Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Bojan Merela

Razvojno okolje za uporabniške vmesnike v vgrajenih sistemih

diplomsko delo

visokošolski strokovni študijski program
računalništvo in informatika

Mentor: viš. pred. dr. Robert Rozman

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Uporabniški vmesniki so običajno zelo pomemben del vgrajenih sistemov. Večina vseh tovrstnih sistemov namreč vzpostavi komunikacijo z uporabnikom; ta mora biti nadvse preprosta in učinkovita. Za obstoječi produkt šifrirne naprave zasnujte in implementirajte optimalno rešitev za izdelavo uporabniškega vmesnika. Pri tem upoštevajte vse posebnosti danega produkta, vključno s posebnostmi vnosnih in prikazovalnih naprav. Problem rešite s širše perspektive problema optimalnega razvoja uporabniških vmesnikov za vgrajene naprave. Preverite in primerjajte najprej obstoječa razvojna okolja in jih po potrebi dopolnite ali nadomestite z lastnimi rešitvami. Celotno rešitev za razvoj uporabniških vmesnikov še uporabite in ovrednotite na danem konkretnem primeru vgrajenega sistema.

Izjava o avtorstvu zaključnega dela

Spodaj podpisani Bojan Merela, vpisna številka 63990272, avtor zaključnega dela z naslovom:

Razvojno okolje za uporabniške vmesnike v vgrajenih sistemih
(angl. Development environment for user interfaces in embedded systems)

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom viš. pred. dr. Roberta Rozmana;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil soglasje etične komisije;
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;
7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V Ljubljani, dne 15. junij 2016 Podpis študenta:

Zahvaljujem se svojemu mentorju viš. pred. dr. Robertu Rozmanu za strokovno in nesebično pomoč pri izdelavi diplomske naloge.

”Mirno morje še nikdar ni naredilo dobrega mornarja!”

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CPU	central processing unit	centralna procesna enota
GPIO	general purpose input/output	splošnonamenski vhod-izhod
GUI	graphical user interface	grafični uporabniški vmesnik
I2C	inter-integrated circuit	povezava med integriranimi vezji
LCD	liquid crystal display	prikazovalnik s tekočimi kristali
LED	light-emitting diode	svetleče diode
OLED	organic light-emitting diode	organske svetleče diode
RGB	red, green, blue	rdeča, zelena, modra
SoC	system on chip	sistem na čipu
SPI	serial peripheral interface	serijski periferni vmesnik

Povzetek

Naslov: Razvojno okolje za uporabniške vmesnike v vgrajenih sistemih

Vgrajeni sistemi so naprave namenjene upravljanju, nadzorovanju ali podpori delovanja nekega večjega sistema, v katerega so vgrajene. Sistemi pogosto ponujajo uporabniške vmesnike, preko katerih jih lahko upravljamo. Diplomsko delo opisuje izdelavo uporabniškega vmesnika za Xiphra SE šifrirne naprave, ki jih izdeluje in trži podjetje Beyond Semiconductor. V prvem delu diplomske naloge se spoznamo z osnovnimi vhodno-izhodnimi napravami in z načinom njihove uporabe znotraj sistema Linux. Na kratko opišemo tudi metode glajenja pisav, preko katerih lahko izboljšamo berljivost uporabnikom prikazanih informacij. Osrednji del govori o načrtovanju uporabniških vmesnikov s poudarkom na razumevanju vgrajenih sistemov ter dodatnih omejitvah, ki jih prinašajo. Vsebuje tudi izdelan načrt razvoja uporabniškega vmesnika v katerem so opredeljene vse naše potrebe in zahteve, vključuje pa tudi analizo primernosti obstoječih rešitev, ogrodij in programskih knjižnic, ki so namenjene hitrejšemu razvoju uporabniških vmesnikov za vgrajene sisteme. Sledi izvedbeni del, ki opisuje gradnjo razvojnega okolja, sestavljenega iz sledečih programskih komponent: gonilnika slikovnega medpomnilnika, orodja za pretvorbo obrisnih pisav, simulatorja slikovnega medpomnilnika, grafične programske knjižnice, knjižnice gradnikov ter uporabniškega vmesnika za Xiphra SE šifrirne naprave. V zaključnem delu pa so strnjene misli o doseženem ter podane možnosti za nadaljni razvoj.

Ključne besede: vgrajen sistem, vhodno-izhodne naprave, uporabniški vmesnik, Linux, glajenje pisav, razvojno okolje, knjižnica gradnikov.

Abstract

Title: Development environment for user interfaces in embedded systems

Embedded systems are devices used to control, monitor or assist the operation of a larger system they are embedded into. They are often controlled through the corresponding user interface. This thesis describes the development of a user interface for the Xiphra SE encryption devices which are produced and marketed by Beyond Semiconductor company. In the first part, the reader will be introduced to a handful of basic input/output devices and with the way they can be used inside the Linux system. Font smoothing methods are explained further on since they are quite often being used to increase readability of the displayed information. The mid-part talks about planning the development of user interfaces influenced by the understanding of specifics and limitations of embedded systems. With this in mind, a plan for the development of the Xiphra SE user interface is created, defining all our needs and requirements. It also evaluates existing solutions: two GUI frameworks and one widget library, which can be used to speed up the development process. In the main part, the thesis describes the creation of the development environment that consists of the following software components: framebuffer driver, font tool, framebuffer simulator, graphics library, widget library and graphical user interface for the Xiphra SE devices. In the last part, we sum up our work and give options for further improvements.

Keywords: embedded system, I/O devices, user interface, Linux, font smoothing, development environment, widget library.

Poglavje 1

Uvod

Vgrajeni sistemi so naprave, ki so namenjene upravljanju, nadzorovanju ali v pomoč nekemu večjemu sistemu v katerega so vgrajene (industrijske naprave, medicinska oprema, avtomobili, telefoni, ...) [8]. Zasnovane so tako, da opravljajo eno ali več točno določenih nalog in so pogosto omejene s strani njihovih snovalcev, ki morajo upoštevati izvedbene, stroškovne in pa tudi časovne zahteve. Pri načrtovanju vgrajenega sistema je tako ključno, da kljub omejitvam obdržimo željeno funkcionalnost [8]. Vgrajeni sistemi pogosto ponujajo uporabniške vmesnike preko katerih jih lahko upravljamo in spreminjamo njihove nastavitve. Slednji predstavljajo most med strojno opremo na eni strani ter uporabnikom na drugi (ang. human-machine interface).

V podjetju Beyond Semiconductor smo pri razvoju lastnih omrežnih šifrirnih naprav družine Xiphra zaznali potrebo po izdelavi uporabniškega vmesnika, preko katerega bi uporabnikom naših naprav omogočili spreminjanje mrežnih nastavitev, spremljanje informacij o delovanju naprave ter jih obveščali o napakah. Po naglem premisleku smo se odločili izdelati tekstovni uporabniški vmesnik s pomočjo programske knjižnice "ncurses". Slednja se je kmalu izkazala za neprimerno, saj ni ponujala vseh potrebnih gradnikov, ki smo jih potrebovali za izdelavo uporabniškega vmesnika. Navkljub vsemu smo manjkajoče gradnike izdelali sami in tako uspešno dokončali uporabniški

vmesnik. Po nekaj tedenski uporabi smo spoznali, da ne omogoča enostavne uporabe naših naprav ter da že s samim izgledom ne bo puščal dobrega vtisa pri uporabnikih. Odločili smo se za ponovno izdelavo, kjer smo uporabnike naših naprav postavili v središče našega načrtovanja in razvojnega procesa.

Zavrgli smo celotno programsko kodo starega uporabniškega vmesnika, novega pa smo nato veliko bolj skrbno načrtovali. Izdelali smo papirnate prototipe s katerimi smo že zelo zgodaj odkrili številne pomankljivosti in jih že v zgodnjih fazah tudi odpravili (stroškovna učinkovitost). Po skrbnem načrtovanju smo tako izdelali povsem nov grafični uporabniški vmesnik.

Cilj te diplomske naloge je prispevati k boljšemu razumevanju vgrajenih uporabniških vmesnikov, bolj premišljenemu načrtovanju ter na koncu predstaviti izdelavo grafičnega uporabniškega vmesnika za Xiphra SE šifrirne naprave. Vse naše ugotovitve želimo deliti z bralcem te diplomske naloge. Želimo mu predstaviti posebnosti vgrajenih sistemov in vsebovane strojne opreme ter pokazati njihov vpliv na zasnovo uporabniškega vmesnika. Prav tako bomo na različnih delih izpostavili pristope, ki smo jih ubrali, da bi uporabnikom omogočili enostavnejšo uporabo našega vmesnika in posledično celotne naprave (uporabniško usmerjeno načrtovanje).

Izdelali smo gonilnik slikovnega medpomnilnika (ang. framebuffer driver) za krmilnik SEPS525 preko katerega lahko v uporabniškem načinu izrisujemo slike na zaslonu slikovnega prikazovalnika. Nato smo izdelali pomožni programski orodji, simulator slikovnega medpomnilnika in pretvornik obrisnih pisav, s katerima smo pohitrili nadaljni razvoj uporabniškega vmesnika. Sledila je izdelava grafične programske knjižnice ("libfbgfx"), ki nam omogoča izris osnovnih geometrijskih oblik, izris slik ter tudi uporabo bitnih pisav za izris teksta. Slednjo smo uporabili za izdelavo programske knjižnice gradnikov uporabniškega vmesnika ("libwidget"), ki je postala jedro našega uporabniškega vmesnika, saj omogoča izris številnih gradnikov (gumb, vnosno polje, slika ...), ki jih potrebujemo za njegovo upodobitev. Z njeno pomočjo smo na koncu izdelali uporabniški vmesnik za Xiphra SE šifrirne naprave.

Uporabniške vmesnike upravljamo oziroma prikazujemo s pomočjo vhodno-izhodnih naprav. Posledično je pomembno poznati njihove lastnosti in omejitve, saj bomo lahko le tako pravilno zasnovali uporabniški vmesnik. Pogledali si jih bomo v poglavju 2, ki opisuje tudi njihovo uporabo znotraj sistema Linux.

Uporabniški vmesnik mora prikazovati informacije jasno, nedvoumno in v primeru teksta tudi berljivo. Ravno berljivost pisav je pogosto okrnjena, saj v primeru uporabe majhnih slikovnih prikazovalnikov nimamo na voljo dovolj slikovnih točk za njihovo lepo upodobitev. Pristope, s katerimi lahko polepšamo upodobitev pisav bomo spoznali v poglavju 3.

Sledi najpomembnejše 4. poglavje v katerem se bomo spoznali z načrtovanjem uporabniških vmesnikov za vgrajene sisteme. Pogledali si bomo kako lahko omejitve programske ali strojne opreme vplivajo na zasnovo našega vmesnika.

V poglavju 5 je predstavljen izdelan načrt, ki vsebuje analizo uporabnikov naših naprav, prototip uporabniškega vmesnika ter pregled vseh naših potreb in zahtev. Poznavanje slednjih nam je služilo pri izbiri obstoječega ogrodja oziroma programske knjižnice (poglavje 5.6) s katero smo si želeli pohitriti izdelavo uporabniškega vmesnika. Žal nismo našli ustrezne rešitve in smo zato podporne programske knjižnice napisali sami.

Izvedbeni del diplomske naloge opisuje sledeče izdelane programske komponente: gonilnik slikovnega medpomnilnika (poglavje 6), simulator slikovnega medpomnilnika (poglavje 7.1), orodje za pretvorbo obrisnih pisav (poglavje 7.2), grafično programsko knjižnico (poglavje 8.1), knjižnico gradnikov uporabniškega vmesnika (poglavje 8.2) ter uporabniški vmesnik za Xiphra SE šifrirne naprave (poglavje 8.3).

V sklepnem delu smo nato povzeli naše delo in podali oceno doseženega. Razmislili smo tudi o možnostih za nadaljni razvoj uporabniškega vmesnika.

Poglavje 2

VI naprave v vgrajenih sistemih Linux

Poznavanje vhodno-izhodnih naprav je pomembno, saj jih med drugim uporabljamo tudi za upravljanje in izrisovanje uporabniških vmesnikov. V nadaljevanju si bomo tako pogledali nekaj izmed takšnih naprav, spoznali pa bomo tudi načine njihove uporabe znotraj operacijskega sistema Linux. Slednji ponuja splošnonamenske vmesnike, preko katerih lahko dostopamo do vhodno-izhodnih naprav. Pogledali si bomo vmesnik vhodnih dogodkov, ki je dostopen preko dogodkovne naprave ("evdev"). Slednji nam omogoča sprejemanje različnih vhodnih dogodkov s strani vhodnih naprav, na primer pritisnjena tipka na tipkovnici. Spoznali bomo tudi vmesnik za dostop do slikovnega medpomnilnika, preko katerega izrisujemo slike na zaslonih slikovnih prikazovalnikov. Dostopen je preko naprave slikovnega medpomnilnika ("fbdev"). Spoznali se bomo tudi s splošnonamenskimi vhodno-izhodnimi nožicami, saj jih pogosto uporabljamo za povezovanje enostavnejših vhodno-izhodnih naprav.

2.1 Splošnonamenske vhodno-izhodne nožice - "GPIO"

Splošnonamenske vhodno-izhodne nožice (ang. GPIO pins) so nožice, ki jih lahko najdemo na integriranih vezjih kot so mikrokontrolerji ter sistemi na čipu (ang. SoC). Lahko so vhodne ali pa izhodne ter nimajo vnaprej določenega pomena. Načrtovalci vgrajenih sistemov jih lahko uporabijo za povezovanje raznih vhodno-izhodnih naprav. Nekaj takšnih vhodnih naprav bomo spoznali v nadaljevanju.

Logične vrednosti nožic lahko nastavljamo in beremo preko krmilnika vhodno-izhodnih nožic. Da jih krmilimo, moramo v vgrajenih sistemih brez operacijskega sistema do krmilnika dostopati neposredno. V nasprotnem primeru pa to delo namesto nas opravi gonilnik splošno namenskih vhodno-izhodnih nožic (ang. GPIO driver), preko katerega potem posredno v uporabniškem načinu upravljamo z nožicami.

2.2 Vhodne naprave

Pogledali si bomo nekaj najpogostejših vhodnih naprav, ki se pojavljajo v vgrajenih sistemih. Osredotočili se bomo predvsem na tiste, preko katerih lahko vgrajeno napravo krmilimo in spreminjamo njene nastavitve oziroma delovanje. Razumevanje vhodnih naprav nam bo pomagalo pri razumevanju razlik v načrtovanju navadnih uporabniških vmesnikov napram tistim namenjenim vgrajenim sistemom.

Modernejši sistemi osnovani na operacijskih sistemih, kot so Linux, Android, Windows ter mnogih drugih, podpirajo tudi vhodne naprave, ki sicer niso tipične za vgrajene sisteme. To so predvsem naprave, ki jih vsakodnevno uporabljamo na osebnih računalnikih: tipkovnice, miške, mikrofoni, optični bralniki, pogojno tudi zasloni na dotik. Tiste, ki jih srečujemo v vgrajenih sistemih, so ponavadi veliko bolj preproste ter skrbno izbrane, saj morajo

opravljati točno določeno vlogo v sistemu in to na učinkovit način. Izbira vhodnih naprav je pogosto ena izmed najpomembnejših odločitev pri načrtovanju vgrajenega sistema, saj pogojuje način, kako bo končni uporabnik to napravo uporabljal. Posledično to vpliva tudi na zasnovo uporabniškega vmesnika v kolikor ga bo naprava imela.

Nekaj vhodnih naprav, ki jih pogosto srečujemo v vgrajenih sistemih:

- tipka
- stikalo
- krožni kodirnik
- zaslon občutljiv na dotik

2.2.1 Tipka

Tipka je ena izmed najbolj pogostih vhodnih naprav. V vgrajenih sistemih je tipično povezana preko splošnonamenske vhodno-izhodne nožice. Ima samo dve stanji: eno, ko je pritisnjena, in drugo, ko ni. Stanje lahko opišemo z enim samim bitom, tipično z '0' kadar ni pritisnjena ter z '1', ko je. V nekaterih sistemih sta lahko stanji tudi obrnjeni. Vsaka tipka ima točno določeno vlogo v sistemu, ki jo določijo snovalci vgrajenega sistema.

Nekaj možnih uporab tipke:

- ponovni zagon naprave
- ponastavljanje nastavitev naprave
- izbira načina delovanja naprave
- upravljanje uporabniškega vmesnika (navigacijska ali potrditvena tipka)

2.2.2 Stikalo

Stikalo se razlikuje od tipke le v tem, da ohranja nastavljeno stanje. Stikala so tipično povezana preko splošnonamenske vhodno-izhodne nožice, v nekaterih primerih pa tudi na posebne nožice sistema na čipu (ang. SoC), preko

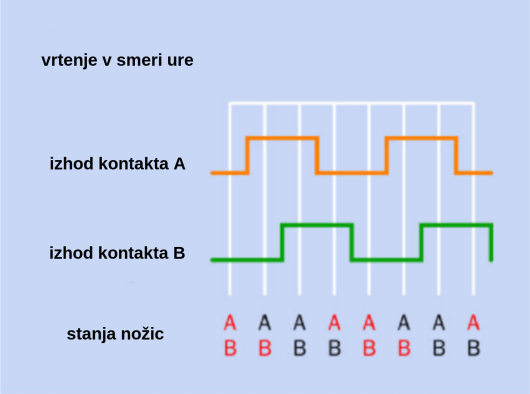
katerih lahko spreminjamo njegove zagonske nastavitve (ang. bootstraps). Uporabniški vmesnik tipično prikazuje le stanja stikal, spremembo delovanja pa sproži sistem sam.

2.2.3 Krožni kodirnik

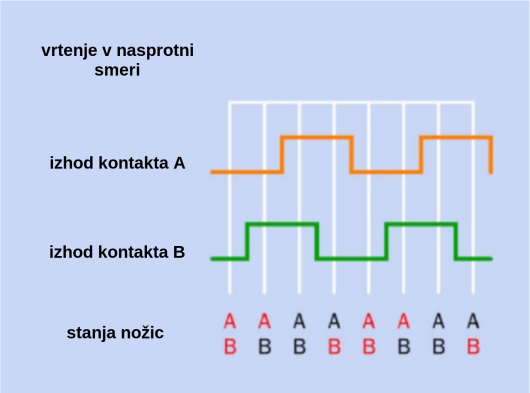
Krožni kodirnik (ang. rotary encoder) je vhodna naprava, ki jo uporabnik lahko zavrti in na tak način na primer upravlja uporabniški vmesnik na LCD zaslonu [3]. Prikazan je na Sliki 2.1. Krožni kodirniki so pogosto narejeni tako, da omogočajo tudi lastnosti navadne tipke. To nam ponuja tri različne vhodne dogodke: premik v levo, premik v desno ter pritisk gumba. Ob premikanju ne hrani svojega stanja, zato v nobenem trenutku ne moremo poznati njegovega absolutnega položaja.

Slika 2.1: Krožni kodirnik [22]

Krožni kodirnik je sestavljen iz dveh parov kontaktov A in B, ki se s premikanjem gumba skleneta in razkleneta v zelo kratkem zamiku. To razliko v fazi lahko zaznamo ter s tem določimo smer premika. V kolikor zaznamo, da se je kontakt A sklenil pred kontaktom B potem smo zaznali premik v desno (Slika 2.2). V nasprotnem primeru pa smo zaznali premik v levo (Slika 2.3).



Slika 2.2: Signala kontaktov A in B v primeru vrtenja krožnega kodirnika v smeri urinega kazalca [3]



Slika 2.3: Signala kontaktov A in B v primeru vrtenja krožnega kodirnika v nasprotni smeri urinega kazalca [3]

Ob vrtenju gumba prihaja do motečih prehodnih pojavov, ki lahko povzročijo sprožitev lažnih dogodkov. Prehodne pojave moramo filtrirati, kar nekateri krmilniki GPIO že podpirajo, toda velikokrat ne uspejo odstraniti vseh. To težavo lahko dokončno rešimo le z dodatnim elektronskim vezjem ali uporabo kakšnega drugega krmnilnika namesto GPIO.

Nekaj možnih uporab krožnih kodirnikov:

- upravljanje grafičnega ali tekstovnega uporabniškega vmesnika
- povečevanje in zmanjševanje veličin kot je npr. jakost zvoka

2.2.4 Zasloni občutljivi na dotik

Zasloni občutljivi na dotik so postali nepogrešljiva vhodno-izhodna naprava mnogih modernejših vgrajenih sistemov (telefoni, medicinska oprema, navigacijska oprema ...). Korenito so spremenili način interakcije uporabnikov z napravami, posledično pa tudi uporabniške vmesnike. Zasloni na dotik so zasnovani z na dotik občutljivo stekleno površino, ki je lahko izdelana s pomočjo različnih tehnologij: uporovne, kapacitivne, svetlobne ter tehnologije zvočnega valovanja.

Uporovne so narejene s pomočjo dveh slojev, ki se ob dotiku skleneta in povzročita spremembo električne upornosti na področju dotika. Z njimi je možno izmeriti moč pritiska, imajo pa slabost, da ne delujejo dobro v primeru večprstnega upravljanja (če sploh).

Kapacitivni senzorji so prekriti z materialom, ki lahko shrani električno energijo. Ko se uporabnik dotakne površine se električni naboj s tega področja prenese na uporabnika.

Tehnologija zvočnega valovanja deluje tako, da se na površini zaslona ustvarjajo ultrazvočni valovi, ki se na mestu dotika delno absorbirajo.

V primeru svetlobne tehnologije pa se uporabljajo svetlobni senzorji (tipično kamere), ki zaznajo odboj infrardečih valov na mestu dotika. Slednji se, zaradi dotika, odbijejo od površine zaslona pod kotom 90 stopinj. Njihova slabost je slaba odzivnost. Vse zgoraj naštete pojave je možno izmeriti in tako zaznati mesto dotika [5].

2.3 Izhodne prikazovalne naprave

Poznavanje prikazovalnih naprav je zelo pomembno, saj pomembno vplivajo na zasnovo uporabniškega vmesnika. Pogosto se srečujemo s takšnimi, ki so tako ali drugače omejene. Te omejitve moramo dobro poznati ter prilagoditi uporabniški vmesnik tako, da bo v popolnosti izkoriščal njihove lastnosti. Do omejitev prihaja predvsem zaradi izbire prikazovalne naprave s strani snovalcev vgrajenih sistemov, saj morajo pri izbiri upoštevati številne kriterije:

- namen vgrajenega sistema
- prostorske omejitve vgrajenega sistema
- omejen nabor podatkovnih vodil (SPI, I2C, PCIe, ...)
- stroškovne omejitve

Prikazovalne naprave so lahko slikovne ali pa tekstovne. Na slednjih lahko prikazujemo le tekstovne uporabniške vmesnike, kot to prikazuje Slika 2.4. So prostrosko omejene, saj so tipično le nekajvrstične (npr. 2, 3 oziroma 4). Namesto črk pa lahko na slikovnih prikazovalnikih spreminjamo vsako posamezno točko na zaslonu. Številu slikovnih točk, ki jih lahko prikažemo v horizontalni oziroma vertikalni smeri na zaslonu, pravimo zaslonska ločljivost. Pišemo jo v sledeči obliki: (št. horizontalnih točk) x (št. vertikalnih točk); npr. 160x128.

Slika 2.4: Primer štirivrstičnega tekstovnega prikazovalnika

Slikovne prikazovalne naprave so namenjene prikazu grafičnih vsebin, med drugim tudi za uporabniške vmesnike vgrajenih naprav. Medsebojno se razlikujejo predvsem po velikosti, barvni globini, podprtih barvnih modelih in

tehnologiji izdelave (LCD, OLED ...). Barvna globina je število različnih barvnih odtenkov, ki jih je slikovna prikazovalna naprava sposobna prikazati. Izražena je v bitih. Barvni model predstavlja organizacijo posamezne slikovne točke oziroma njeno predstavitev v pomnilniku prikazovalne naprave. Pogosto lahko izbiramo med različnimi barvnimi modeli na isti napravi. Najbolj pogosti so: monokromatski, sivinski ter RGB.

2.3.1 Monokromatski barvni model

Monokromatski (ang. monochrome) barvni model opisuje vsako slikovno točko z le enim bitom. Tipično '0' za neaktivno stanje ter '1' za aktivno (lahko tudi obratno). Barva slikovne točke v aktivnem stanju je vnaprej določena. Tipično imamo opravka s tako imenovanimi črno-belimi zasloni, kakršnega prikazuje Slika 2.5. Monokromatski prikazovalniki so cenejši od barvnih in se posledično še vedno veliko uporabljajo v vgrajenih sistemih. Zavedati se moramo tudi, da na njih ne bomo mogli uporabljati metod glajenja teksta, ki jih bomo spoznali v enem izmed naslednjih poglavij.

Slika 2.5: Monokromatski slikovni prikazovalnik (Adafruit) [14]

2.3.2 Sivinski barvni model

Sivinski barvni model podaja intenziteto vnaprej določenega barvnega odtenka, najpogosteje bele barve. Intenziteta je sicer lahko predstavljena s poljubnim številom bitov, toda tipične vrednosti so ponavadi: 2, 4 ali 8 bitov (4, 16 ter 256 sivinskih odtenkov). Slika 2.6 prikazuje primer 4-bitnega sivinskega slikovnega prikazovalnika oziroma zaslona.

Slika 2.6: 4-bitni sivinski slikovni prikazovalnik (Crystalfontz) [16]

2.3.3 Barvni model RGB

Barvni model RGB opisuje vsako točko z intenziteto rdeče (R), zelene (G) ter modre (B) barvne komponente. Te barve so osnovne, saj lahko z njihovim mešanjem pridobimo vse ostale. Vsaka izmed njih je predstavljena z določenim številom bitov, ki pa je odvisno od barvne globine slikovnega prikazovalnika. Slika 2.7 prikazuje primer 16-bitnega slikovnega prikazovalnika RGB.

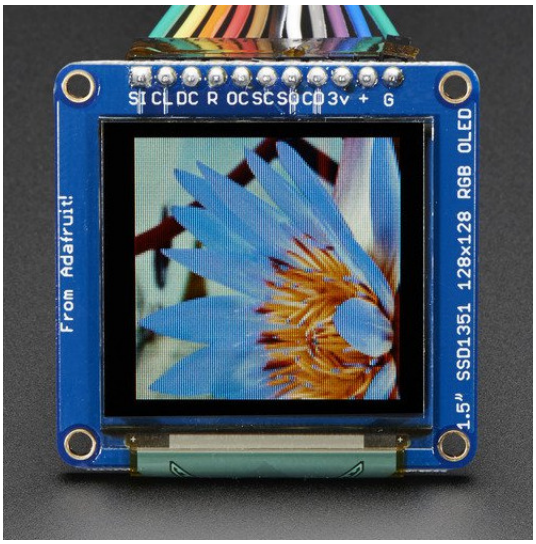
Barvnih modelov RGB, katerih barvna globina ni deljiva z osem, se pogosto izogibamo, saj moramo pri njih računati bitne odmike slikovnih točk znotraj slikovnega medpomnilnika. Prav tako barvni model RGBA (Tabela 2.1) ne prinaša nobene bistvene prednosti, saj z njim ne moremo opisati večjega števila barv kot pri modelu RGB. Prinaša le dodatno komponento 'A', s katero je podana transparentnost barvnega odtenka, ki pa se za uporabniške

vmesnike na vgrajenih sistemih tipično ne uporablja. Izjema bi lahko bile medicinske naprave, na katerih je pogosto potrebno prikazati kompozicijo različnih slik, kjer so zgornje slike transparentne glede na tiste v ozadju.

Poznamo sledeče tipe barvnih modelov RGB:

barvni model	**barvna globina**	**rdeča (R)**	**zelena (G)**	**modra (B)**	**prozornost (A)**
RGB444:	12-bitna	4 biti	4 biti	4 biti	/
RGB565:	16-bitna	5 bitov	6 bitov	5 bitov	/
RGB666:	18-bitna	6 bitov	6 bitov	6 bitov	/
RGB888:	24-bitna	8 bitov	8 bitov	8 bitov	/
RGBA:	32-bitna	8 bitov	8 bitov	8 bitov	8 bitov

Tabela 2.1: Pregled barvnih modelov RGB



Slika 2.7: 16-bitni slikovni prikazovalnik RGB (Adafruit) [14]

2.4 Upravljanje splošnonamenskih nožic

Znotraj operacijskega sistema Linux je delo s splošnonamenskimi nožicami sila enostavno. Od gonilnika GPIO lahko zahtevamo, da preko sistemske datoteke izvozi stanje posamezne nožice v uporabniški način. Zahteva se sproži tako, da vpišemo zaporedno številko nožice v sistemsko datoteko: /sys/class/gpio/export. Po vpisu bo gonilnik ustvaril sistemsko mapo: /sys/class/gpio/gpioX, kjer črka X predstavlja zaporedno številko nožice. Znotraj te mape se bodo ustvarile tudi sistemske datoteke, preko katerih jih bomo lahko upravljali. Sistemska datoteka "direction"upravlja s smerjo nožice. V kolikor vanjo zapišemo znak '0', bo ta postala vhodna, če pa vanjo zapišemo znak '1', pa bo postala izhodna. Na koncu moramo preko sistemske datoteke "value"le še nastaviti logično vrednost nožice. Vpišemo lahko znaka '0' ali '1'. Programska koda lahko z uporabo datotečnih sistemskih klicev "open()", "read()" in "write()", upravlja z zgoraj omenjenimi datotekami. Celoten postopek znotraj ukazne lupine Linux prikazuje Slika 2.8.

```
echo 1 > /sys/class/gpio/export
echo 0 > /sys/class/gpio/gpio1/direction
cat /sys/class/gpio/gpio1/value
```

Slika 2.8: Ugotavljanje stanja GPIO nožice št. 1, znotraj ukazne lupine Linux.

Tipične vhodne naprave povezane preko GPIO nožic:

- tipka
- stikalo

Tipične izhodne naprave povezane preko GPIO nožic:

- LED diode

2.5 Vmesnik vhodnih dogodkov - "evdev"

Vmesnik vhodnih dogodkov (ang. "evdev") je splošnonamenski vhodni vmesnik, preko katerega lahko v uporabniškem načinu pridobivamo vhodne dogodke. Vmesnik je zasnovan tako, da podpira širok nabor naprav. Preko njega lahko uporabniški programi pridobijo dogodke, ki se pojavijo na napravah kot so miške, tipkovnice, krožni kodirniki, zasloni občutljivi na dotik in podobno. Jedro Linux ustvari za vsako vhodno napravo, ki deluje preko vmesnika vhodnih dogodkov, svojo sistemsko datoteko: /dev/input/eventX, kjer črka X predstavlja enoznačno identifikacijsko številko naprave (primer: /dev/input/event0, /dev/input/event1 ...). Uporabniški program mora nato le še odpreti sistemsko datoteko izbrane naprave ter iz nje brati. Z branjem pridobljeni podatki so v obliki podatkovne strukture "input_event", ki si jo bomo pogledali v nadaljevanju.

Še prej pa si poglejmo celotno kronološko pot vhodnega dogodka:

- Prišlo je do dogodka na vhodni napravi, npr.: pritisnjena tipka na tipkovnici.
- Gonilnik vhodne naprave je zaznal pritisnjeno tipko ter o tem obvestil jedro Linux preko vmesnika vhodnih dogodkov.
- Vmesnik vhodnih dogodkov je shranil dogodek in čaka, da ga bo uporabniški program prebral.
- Uporabniški program prebere ustrezno sistemsko datoteko ter s tem pridobi vse informacije o dogodku. Dogodek je bil poslan in sprejet kot podatkovna struktura "input_event".
- Uporabniški program uporabi dogodek.

Podatkovna struktura "input_dev"[11] vsebuje štiri elemente:

- **time** - čas v katerem je bil dogodek ustvarjen (časovna oznaka)
- **type** - tip dogodka

- **code** - enoznačna oznaka dogodka
- **value** - vrednost, ki je pripeta dogodku

Časovna oznaka:

Časovna oznaka (ang. timestamp) je pripeta vsakemu dogodku. Uporabniški vmesnik jo lahko uporabi za izračun razlike v času med dvema dogodkoma, slednjo pa uporabi za pospešitev premikanja po njenih gradnikih. Predpostavimo, da uporabljamo krožni kodirnik. Ob zaznavi večih dogodkov v zelo kratkem času bi se na primer lahko odločili za preskok kakšnega izmed gradnikov uporabniškega vmesnika ter tako pospešili navigacijo.

Tip dogodka:

Vsaka vhodna naprava, ki deluje preko vmesnika vhodnih dogodkov, sporoča enega ali več tipov vhodnih dogodkov. Računalniška miška na primer sporoča svoje relativne premike ter tudi dogodke ob pritisku katerega izmed njenih tipk.

Tip dogodka lahko zavzame sledeče vrednosti:

- **EV_ABS** - absolutni položaj. Ena izmed naprav, ki nam sporoča ta tip dogodka je npr. zaslon občutljiv na dotik.
- **EV_REL** - relativni premik. Ena izmed naprav, ki nam sporoča ta tip dogodka je npr. računalniška miška.
- **EV_KEY** - tip dogodka, ki ga je sprožila pritisnjena tipka.

Enoznačna oznaka dogodka:

V kolikor je tip dogodka EV_ABS ali EV_REL, potem nam enoznačna oznaka dogodka (krajše koda) označuje del naprave, ki je zaznal premik. Lahko je to npr. premik po horizontalni osi (ABS_X, REL_X) ali pa po vertikalni

(ABS_Y, REL_Y). V primeru dogodka EV_KEY pa nam koda označuje pritisnjeno tipko (KEY_A, KEY_B, KEY_0, KEY_1, KEY_ESC, KEY_ENTER ...). Kod je še veliko več [12].

Vrednost:

V kolikor je tip dogodka EV_ABS, nam vrednost pove absolutni položaj naprave, ki je sprožila dogodek. V primeru dogodka EV_REL nam vrednost pove za koliko enot in v kateri smeri se je zgodil premik. Kadar pa je tip dogodka EV_KEY, nam vrednost pove ali je bila tipka spuščena (0), pritisnjena (1) ali pa se je dogodek ponovil (2) zaradi vklopljenega avtomatskega ponavljanja.

2.5.1 Krožni kodirnik

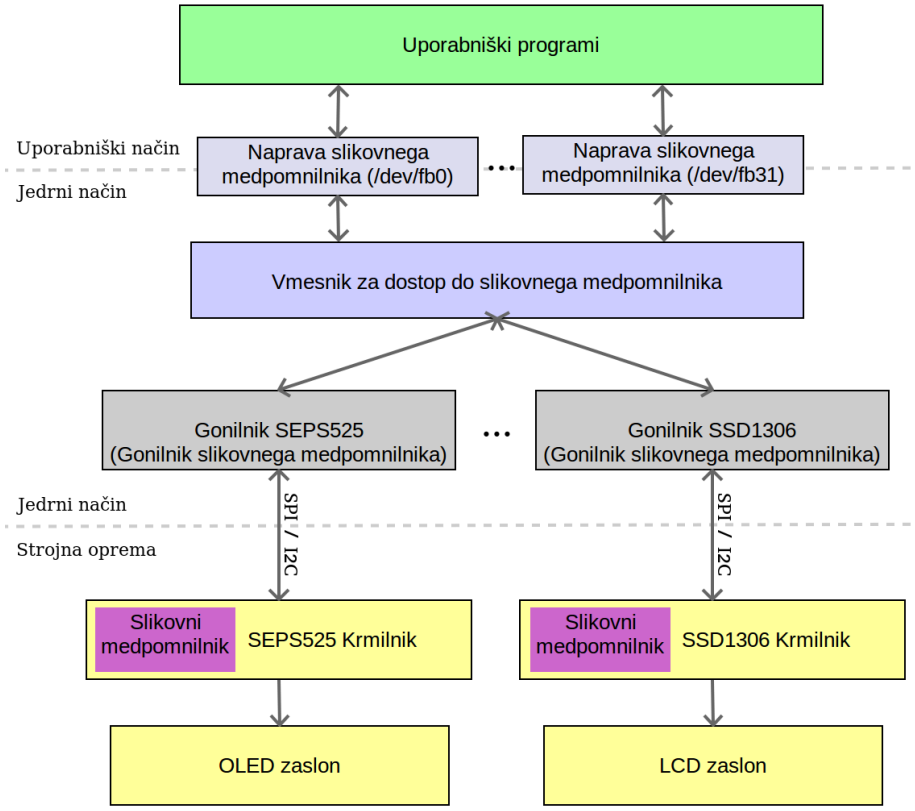
Dogodek krožnega kodirnika je uporabniškemu programu dostopen preko dogodkovne naprave. Gonilnik krožnega kodirnika pošilja relativni dogodek (EV_REL) ter število premikov, ki jih je zaznal. Pozitivno število pomeni vrtenje v desno, negativno pa vrtenje v levo. Uporabniški program bi načeloma lahko tudi sam dekodiral smer vrtenja z branjem ustreznih GPIO nožic, toda takšna rešitev ne bi bila optimalna, saj v uporabniškem načinu nimamo na voljo prekinitev. Slednje nam namreč omogočajo zaznavo dogodka, ko se le ta zgodi in nam zato ni potrebno neprestano spremljati nožic.

2.5.2 Zasloni občutljivi na dotik

Gonilniki zaslonov, občutljivih na dotik, nas preko dogodkovnega vmesnika obveščajo o dogodkih. Pošiljajo nam absolutni (EV_ABS) položaj dotika prsta. V kolikor je možno z zaslonom upravljati večprstno, pa dobimo za vsak dotik po en dogodek, ki je še dodatno opremljen z enoznačno oznako za sledenje.

2.6 Slikovni medpomnilnik - ”framebuffer”

Slikovni medpomnilnik (ang. ”framebuffer”) je del pomnilnika, ki se tipično nahaja v krmilnikih slikovnih prikazovalnikov (grafične kartice, krmilnik SSD1306, krmilnik SEPS525 ...) [4]. Namenjen je hranjenju slike, katero krmilnik neprestano izrisuje na zaslon. V sistemu Linux lahko dostopamo do slikovnega medpomnilnika preko ustrezne naprave slikovnega medpomnilnika (”fbdev”), ki določa abstrakcijski nivo med pogosto zelo različno grafično strojno opremo in uporabniškimi programi. Slednjim ponuja zelo dober, vnaprej zasnovan vmesnik za dostop do slikovnega medpomnilnika (Slika 2.9).



Slika 2.9: Dostop do slikovnega medpomnilnika preko ustrezne naprave Linux sistema

Abstrakcijski nivo določa sledeče vmesnike:

- vmesnik za dostop do slikovnega medpomnilnika - "fbdev"
- vmesnik za poizvedbo in spreminjanje nastavitev [19]

2.6.1 Vmesnik za dostop do slikovnega medpomnilnika

Vmesnik ustvari za vsak gonilnik po eno napravo slikovnega medpomnilnika. Najdemo jih lahko v sistemski mapi /dev kot znakovne naprave (ang. character device): /dev/fb[0-31] [4]. Ker operacijski sistem Linux obravnava vse znakovne naprave kot čisto navadne datoteke, lahko tudi pri napravi slikovnega medpomnilnika uporabimo vse standardne sistemske klice za delo z datotekami: open(), close(), seek(), read(), write() in mmap().

Napravo slikovnega medpomnilnika moramo pred uporabo odpreti s pomočjo sistemskega klica **open()**. V kolikor ne pride do napake pri odpiranju, bomo lahko nad njo uporabili sistemska klica za branje in pisanje: **read()** in **write()**. S pisanjem v napravo posredno prek gonilnika dostopamo do slikovnega medpomnilnika in na tak način izrišemo slike na zaslonu slikovnega prikazovalnika. Enako je pri branju, le da se podatki oziroma slike prenašajo v nasprotni smeri. Ko naprave ne potrebujemo več, jo moramo zapreti z uporabo sistemskega klica **close()**.

Sistemski klic **seek()** uporabimo takrat, ko želimo prebrati oziroma prepisati le del slikovnega medpomnilnika. Z njim lahko nastavimo datotečni odmik oziroma v našem primeru odmik znotraj slikovnega medpomnilnika. Odmik se uporabi šele ob naslednjem branju oziroma pisanju v napravo slikovnega medpomnilnika. Sistemski klic **mmap()** pa nam omogoča preslikavo znakovne naprave oziroma datoteke v pomnilnik uporabniškega procesa. Vse nadaljne operacije za dostop do slikovnega medpomnilnika so nato le še pomnilniško pisanje oziroma branje v to mapirano področje. Tipično ga uporabimo takrat, ko potrebujemo pogosto osveževati le del slikovnega medpomnilnika, kot je to npr. potrebno pri utripanju tekstovnega kurzorja.

Enako bi lahko dosegli z uporabo datotečnih operacij, toda te so časovno zahtevnejše, saj bi morali uporabiti dva zaporedna sistemska klica enega za drugim: seek() ter read() oziroma write(). Uporabo mmap() sistemskega klica prikazuje Slika 2.10.

```
fd = open("/dev/fb0", O_RDWR);

fbmem = mmap(NULL, size, PROT_READ | PROT_WRITE,
             MAP_SHARED, fd, 0);

color = fbmem[0];       /* read from the frame buffer */
fbmem[1] = color;       /* write to the frame buffer */

close(fd);
```

Slika 2.10: Uporaba sistemskega klica mmap nad napravo slikovnega medpomnilnika

2.6.2 Vmesnik za poizvedbo in nastavitve

Vmesnik je zasnovan na podlagi sistemskih klicev "ioctl" ter podatkovnih strukturah, s katerimi je možno opisati širok nabor slikovnih prikazovalnih naprav. Preko njega lahko uporabniški programi pridobijo informacije o sami napravi in njenih nastavitvah (zaslonska ločljivost, barvna globina, hitrost osveževanja ...). Poizvedbe in spremembe se pošiljajo in sprejemajo preko ioctl sistemskega klica, ki je eden izmed načinov komunikacije med uporabniškimi procesi ter gonilniki. Pridobimo lahko dinamične in statične informacije. Slednje opisujejo samo napravo in se ne spreminjajo, medtem ko so dinamične nastavljive in opisujejo trenutni način, v katerem se naprava nahaja. Pred uporabo slikovnega medpomnilnika moramo poslati poizvedbo tako za statične kot dinamične informacije, saj ga bomo lahko le tako pravilno uporabljali. Podatkovni strukturi, ki hranita statične in dinamične informacije, si bomo pobližje pogledali v nadaljevanju.

2.6.3 Podatkovna struktura za statične informacije

Pridobimo jih lahko preko sistemskega klica "ioctl", ki mu moramo podati argument FBIOGET_FSCREENINFO, kot je to razvidno iz Slike 2.11. Naprava slikovnega medpomnilnika nam v primeru uspešno izvršenega sistemskega klica vrne statične informacije preko podatkovne strukture "fb_fix_screeninfo", ki jo prikazuje Slika 2.12.

```
struct fb_fix_screeninfo fb_fix_info;
ioctl(fd, FBIOGET_FSCREENINFO, &fb_fix_info);
```

Slika 2.11: Sistemski klic za pridobivanje statičnih informacij o napravi

```
struct fb_fix_screeninfo {
   char id[16];              /* identification string        */
   unsigned long smem_start; /* Start of frame buffer mem    */
                             /* (physical address)           */
   __u32 smem_len;           /* Length of frame buffer mem   */
   __u32 type;               /* see FB_TYPE_*                */
   __u32 type_aux;           /* Interleave                   */
   __u32 visual;             /* see FB_VISUAL_*              */
   __u16 xpanstep;           /* zero if no hardware panning  */
   __u16 ypanstep;           /* zero if no hardware panning  */
   __u16 ywrapstep;          /* zero if no hardware ywrap    */
   __u32 line_length;        /* length of a line in bytes    */
   unsigned long mmio_start; /* Start of Memory Mapped I/O   */
                             /* (physical address)           */
   __u32 mmio_len;           /* Length of Memory Mapped I/O  */
   __u32 accel;              /* Indicate to driver which     */
                             /*  specific chip/card we have  */
   __u16 capabilities;       /* see FB_CAP_*                 */
};
```

Slika 2.12: Prikaz podatkovne strukture s statičnimi informacijami o slikovnem medpomnilniku

V nadaljevanju bomo naredili pregled elementov podatkovne strukture "fb_fix_screeninfo", ki jo prikazuje Slika 2.12.

Element id:

Element id podaja enoznačno ime naprave slikovnega medpomnilnika.

Element smem_start:

Fizični naslov dela pomnilnika, ki ga gonilnik slikovnega medpomnilnika uporablja za izris slike. Pravzaprav je to dodatni slikovni medpomnilnik znotraj gonilnika, ki se preko različnih podatkovnih vodil (I2C, SPI, PCIe ...) prenaša v medpomnilnik slikovne prikazovalne naprave. Pravimo mu lahko tudi navidezni slikovni medpomnilnik. Fizični naslov lahko uporabimo v uporabniškem načinu tako, da ga preko naprave /dev/mem in z uporabo sistemskega klica mmap preslikamo v navidezni naslovni prostor uporabniškega procesa. Navidezni naslov potem uporabimo za neposreden dostop do navideznega slikovnega medpomnilnika.

Element smem_len:

Podaja dolžino oziroma velikost navideznega slikovnega medpomnilnika, ki vedno ustreza velikosti pravega slikovnega medpomnilnika. Uporabimo ga pri preslikavi fizičnega naslova ter pri številnih slikovnih operacijah.

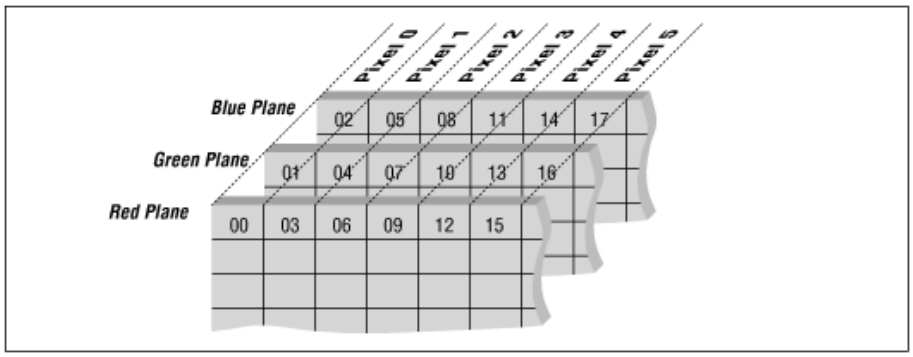
Element type:

Podaja organizacijo slikovnih točk znotraj navideznega slikovnega medpomnilnika. Ta tipično ustreza organizaciji slikovnih točk pravega slikovnega medpomnilnika, ni pa to nujno. V slednjem primeru mora gonilnik slikovnega medpomnilnika pred izrisovanjem narediti pretvorbo. Možne organizacije slikovnih točk:

- **FB_TYPE_PACKED_PIXELS** - Slikovne točke so shranjene zaporedno v eni sami ravnini in so lahko bitno skrčene skupaj v primeru, ko

posamezna točka zaseda število bitov, ki ni deljivo z osem. V nasprotnem primeru pa se vsaka točka razširi tako, da zasede celoštevilsko število bajtov. Da ugotovimo, ali so točke skrčene ali ne, moramo preveriti tudi element visual.

- **FB_TYPE_PLANES** - Slikovne točke so shranjene v več ravninah. Vsak bit posamezne točke pa se nahaja v svoji ravnini. Ravnine so v pomnilniku shranjene zaporedoma. Slika 2.13 prikazuje primer takšne organizacije slikovnih točk.
- **FB_TYPE_INTERLEAVED_PLANES** - Slikovne točke so shranjene v več ravninah. Vsak bit točke pa se nahaja v svoji ravnini. Ravnine so v pomnilniku prepletene (ang. interleaved).



Slika 2.13: Barvne ravnine RGB kjer je vsak bit slikovne točke shranjen v svoji ravnini [2]

V novejših vgrajenih sistemih se bomo najpogosteje srečali z zaporedno organizacijo slikovnih točk v eni sami ravnini (FB_TYPE_PACKED_PIXELS).

Element visual:

Določa barvni model slikovnih točk. Navidezni slikovni medpomnilnik podpira sledeče barvne modele:

- **FB_VISUAL_MONO01** - monokromatski barvni model (črno-bel). Slikovne točke so shranjene v pomnilniku kot skupek bitov, ponavadi en sam bit. Za predstavitev črne barve moramo vse bite slikovne točke postaviti na 1, za belo barvo pa na 0. Več točk je lahko bitno skrčenih znotraj enega bajta, v kolikor je število potrebnih bitov za predstavitev ene točke manjše od osem.

- **FB_VISUAL_MONO10** - monokromatski barvni model (črno-bel). Slikovne točke so shranjene v pomnilniku kot skupek bitov, ponavadi en sam bit. Za predstavitev črne barve moramo vse bite slikovne točke postaviti na 0, za belo barvo pa na 1 (ravno obratno kot pri barvnem modelu FB_VISUAL_MONO01). Več točk je lahko bitno skrčenih znotraj enega bajta, v kolikor je število potrebnih bitov za predstavitev ene točke manjše od osem.

- **FB_VISUAL_PSEUDOCOLOR** - slikovne točke so predstavljene kot odmik v barvni tabeli, v kateri so shranjene rdeče, zelene in modre barvne komponente.

- **FB_VISUAL_TRUECOLOR** - RGB barvni model. Vsaka slikovna točka je razdeljena v tri barvne komponente: rdečo (R), zeleno (G) in modro (B). Vsaka izmed komponent pa predstavlja odmik znotraj iskalne tabele, katero je možno samo brati in je pogojena s strani strojne opreme. Slednja predstavlja barvno paleto, ki jo krmilnik slikovne prikazovalne naprave uporablja za izris slikovnih točk. Število bitov potrebnih za predstavitev posamezne komponente je lahko različno (RGB565, RGB666, RGB888 ...).

- **FB_VISUAL_DIRECTCOLOR** - RGB barvni model. Vsaka slikovna točka je razdeljena v tri barvne komponente: rdečo (R), zeleno (G) in modro (B). Vsaka izmed komponent pa predstavlja odmik znotraj iskalne tabele, v katero je možno tudi pisati. Slednja predstavlja barvno paleto, ki jo krmilnik slikovne prikazovalne naprave uporablja za izris slikovnih točk.

Elementi xpanstep, ypanstep in ywrapstep:

Nekateri krmilniki slikovnih prikazovalnih naprav podpirajo strojno premikanje slike (ang. scrolling). To deluje tako, da se na zaslonu izrisuje le izbran del slikovnega medpomnilnika. Temu delu rečemo prikazovalno okno (ang. view window). Elementa podatkovne strukture **xpanstep** in **ypanstep** tako določata, za koliko slikovnih točk naj se prikazovalno okno premakne v vsakem koraku. Element **ywrapstep** ima enak učinek kot ypanstep toda omogoča hitrejše premikanje.

Element line_length:

Določa dolžino ene zaslonske vrstice v bajtih. Podatek lahko uporabimo pri računanju pomnilniških naslovov slikovnih točk, kot to prikazuje enačba 2.1.

$$address = y * line\_length + x * bytes\_per\_pixel \tag{2.1}$$

Elementa mmio_start in mmio_len:

Določata naslov in dolžino pomnilniško preslikanega vhoda-izhoda.

2.6.4 Podatkovna struktura za dinamične informacije

Pridobimo jih lahko preko sistemskega klica "ioctl", ki mu moramo podati argument FBIOGET_VSCREENINFO oziroma FBIOPUT_VSCREENINFO kadar želimo nastavljati. Oba sistemska klica sta prikazana na Sliki 2.14. Naprava slikovnega medpomnilnika nam v primeru uspešno izvršenega sistemskega klica vrne dinamične informacije preko podatkovne strukture "fb_var_screeninfo", ki jo prikazuje Slika 2.15.

```
struct fb_var_screeninfo fb_var_info;

ioctl(fd, FBIOGET_VSCREENINFO, &fb_var_info);

fb_var_info.xres = 640;
fb_var_info.yres = 480;

ioctl(fd, FBIOPUT_VSCREENINFO, &fb_var_info);
```

Slika 2.14: Sistemska klica za pridobivanje oziroma nastavljanje dinamičnih nastavitev naprave

```
struct fb_var_screeninfo {
   __u32 xres;                 /* visible resolution         */
   __u32 yres;
   __u32 xres_virtual;         /* virtual resolution         */
   __u32 yres_virtual;
   __u32 xoffset;              /* offset from virtual to     */
   __u32 yoffset;              /* visible resolution         */
   __u32 bits_per_pixel;       /* guess what                 */
   __u32 grayscale;            /* 0 = color, 1 = grayscale,  */
   struct fb_bitfield red;     /* bitfield in fb mem if true */
   struct fb_bitfield green;   /* color, else only length is */
   struct fb_bitfield blue;    /* significant                */
   struct fb_bitfield transp;  /* transparency               */
   __u32 nonstd;               /* != 0 Non std. pixel format */
   __u32 activate;             /* see FB_ACTIVATE_*          */
   __u32 height;               /* height of picture in mm    */
   __u32 width;                /* width of picture in mm     */

   ...
};
```

Slika 2.15: Prikaz podatkovne strukture z dinamičnimi informacijami o slikovnem medpomnilniku

V nadaljevanju bomo naredili pregled elementov podatkovne strukture "fb_var_screeninfo", ki jo prikazuje Slika 2.15.

Elementa xres in yres:

Podajata zaslonsko ločljivost v horizontalni in vertikalni smeri. Uporabimo ju lahko za izračun velikosti slikovnega medpomnilnika, kot je to razvidno iz spodnjih enačb:

$$size\_bytes = (xres * yres * bits\_per\_pixel)/8 \tag{2.2}$$

$$size\_bytes = yres * line\_length \tag{2.3}$$

Uporabimo ju tudi za omejevanje izrisovalnih algoritmov, kot so npr.: črta, kvadrat, krog in številni drugi. Omejimo jih tako, da ne poskušamo izrisovati izven zaslonske površine.

Elementi xres_virtual, yres_virtual, xoffset in yoffset:

Elementa **xres_virtual** in **yres_virtual** podajata navidezno zaslonsko ločljivost, ki je lahko enaka ali pa večja od ločljivosti zaslona prikazovalne naprave. Tako lahko pišemo v del slikovnega medpomnilnika, ki se trenutno ne izrisuje ter ga prikažemo kasneje s premikom prikazovalnega okna v to področje. Prikazovalno okno lahko premaknemo tako, da nastavimo sledeča elementa podatkovne strukture: **xoffset** in **yoffset**.

Element bits_per_pixel:

Podaja število bitov potrebnih za predstavitev posamezne slikovne točke oziroma njeno barvno globino.

Element grayscale:

Določa uporabo sivinskega barvnega modela (vrednost 1). Ko slednji ni nastavljen (vrednost 0), pa nam barvni model podajajo sledeči elementi podatkovne strukture: "red", "green", "blue"in "transp".

Elementi red, green, blue in transp:

Podani so s podatkovno strukturo tipa "fb_bitfield", ki jo prikazuje Slika 2.16. Struktura vsebuje informacijo o številu bitov ("length"), potrebnih za predstavitev barvne komponente, njen odmik znotraj barvnega modela ("offset") ter podatek o tem kateri bit ima največjo težo ("msb_right"). Na ta način lahko opišemo kateri koli barvni model RGB (RGB444, RGB565, RGB666, RGB888, RGBA ...).

```
struct fb_bitfield {
    __u32 offset;     /* beginning of bitfield       */
    __u32 length;     /* length of bitfield          */
    __u32 msb_right;  /* != 0 : Most significant bit */
};
```

Slika 2.16: Podatkovna struktura fb_bitfield

Prikaz uporabe podatkovne strukture "fb_bitfield" za nastavitev barvnega modela RGB565 (Slika 2.17).

```
struct fb_var_screeninfo fb_var_info;

ioctl(fd, FBIOGET_VSCREENINFO, &fb_var_info);

fb_var_info.red.offset      = 0;
fb_var_info.red.length      = 5;
fb_var_info.green.offset    = 5;
fb_var_info.green.length    = 6;
fb_var_info.blue.offset     = 11;
fb_var_info.blue.length     = 5;

ioctl(fd, FBIOPUT_VSCREENINFO, &fb_var_info);
```

Slika 2.17: Nastavitev barvnega modela RGB565

Element nonstd:

V kolikor je vrednost različna od nič, imamo opravka z nestandardnim barvnim modelom slikovne točke.

Element activate:

Pri nastavljanju slikovnega medpomnilnika lahko z nastavitvijo activate določimo, kdaj naj se nove nastavitve uporabijo (Slika 2.18). Nastavimo lahko sledeče:

- **FB_ACTIVATE_NOW** - nastavitve naj se nastavijo takoj
- **FB_ACTIVATE_NXTOPEN** - nastavitve naj se nastavijo ob naslednjem odpiranju okvirnega medpomnilnika
- **FB_ACTIVATE_TEST** - nastavitve naj se ne nastavijo. To nastavitev uporabimo takrat, ko želimo preveriti ali smo vse nastavili pravilno oziroma če okvirni medpomnilnik podpira željene nastavitve.
- **FB_ACTIVATE_VBL** - nastavitve naj se nastavijo tik pred začetkom naslednjega vertikalnega osveževanja
- **FB_ACTIVATE_FORCE** - nastavitve naj se nastavijo, tudi če v njih ni nobene spremembe

```
struct fb_var_screeninfo fb_var_info;

fb_var_info.activate = FB_ACTIVATE_NOW | FB_ACTIVATE_FORCE;

ioctl(fd, FBIOPUT_VSCREENINFO, &fb_var_info);
```

Slika 2.18: Takojšnja nastavitev dinamičnih informacij (tudi, če ni nobene spremembe)

Elementa width in height:

Podajata velikost slike v milimetrih (width podaja širino, height pa višino).

Poglavje 3

Prikaz teksta na slikovnih prikazovalnikih

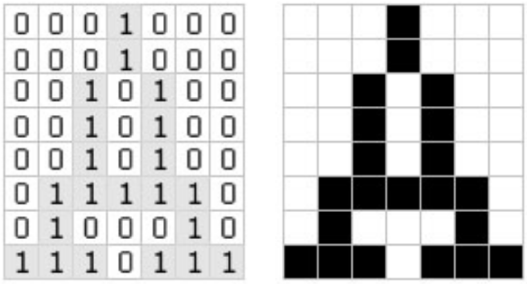
Uporabniku moramo prikazati informacije na jasen, nedvoumen in v primeru teksta na berljiv način. Ravno slednje je na majhnih slikovnih prikazovalnikih pogosto okrnjeno, saj nimamo na voljo dovolj slikovnih točk za lepo upodobitev majhnih pisav. V tem poglavju se bomo spoznali z bitnimi pisavami ter s pristopi za njihovo lepšo upodobitev.

3.1 Bitne pisave

Uporabniški vmesniki na vgrajenih sistemih za prikazovanje teksta najpogosteje uporabljajo tako imenovane bitne pisave (ang. bitmap fonts) [18], saj za njihov prikaz ne potrebujemo veliko procesne moči. Bitna pisava vsebuje bitno sliko za vsak posamezen glif oziroma znak. Za prikaz znaka moramo le izrisati njegovo bitno sliko, kot to prikazuje Slika 3.1.

Bitne slike lahko narišemo ali pa jih pridobimo s pretvorbo vektorskih oziroma obrisnih pisav (ang. outline fonts), ki so zasnovane na podlagi vektorjev in Bézierovih krivulj. Bitne pisave niso prilagodljive, saj moramo za vsako velikost oziroma slog izdelati nove bitne slike [7]. V primeru uporabe večjega števila pisav bomo posledično porabili nekoliko več delovnega

pomnilnika. Prikazani znaki bodo imeli ostre robove (Slika 3.2), kar lahko okrni izgled uporabniškega vmesnika ter poslabša berljivost prikazanih informacij. Težavo je možno omiliti z uporabo ene izmed metod tekstovnega glajenja, ki so predstavljene v nadaljevanju.



Slika 3.1: Binarna slika (levo) in prikaz znaka A (desno) [18].

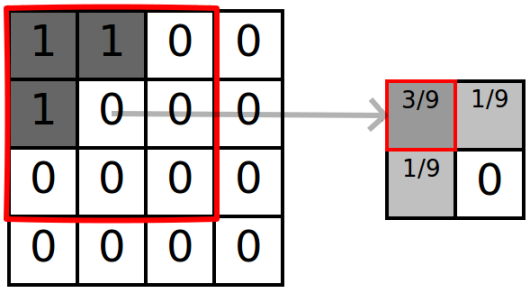
Velikost bitne pisave podajamo s širino in višino vsebovanih znakov. Pišemo jo v obliki: širina x višina (npr.: 5x7, 6x7, 7x11, 17x31 ...).

Slika 3.2: Neglajen znak 'g', prikazan v različnih ločljivostih [13]

3.2 Glajenje teksta z metodo glajenja krivulj

Pri metodi glajenja krivulj (ang. antialiasing) je vsaka slikovna točka znaka predstavljena z intenziteto barvnega odtenka, ki je odvisna od deleža krivulje, ki bi prekrival posamezno slikovno točko [18] (znak je pravzaprav predstavljen kot sivinska slika intenzitet). Najpogostejši pristop je uporaba prevzorčenja (ang. oversampling), kjer znak obrisne pisave upodobimo v nekajkrat večji velikosti (2x, 4x, 8x, ...) in ga nato pomanjšamo z uporabo metode za zmanjševanje vzorčenja (ang. downsampling), npr. povprečne (ang. avarage) ali utežene metode (ang. bicubic). Intenziteta posamezne slikovne

točke znaka je tako povprečje ustreznih sosednjih točk znotraj prevzorčene bitne slike (Slika 3.3). Glajenje teksta s to metodo ni možno v primeru monokromatskega slikovnega prikazovalnika, prav tako ni smiselno ob zelo majhni velikosti pisave. Primer glajenega znaka prikazuje Slika 3.4.

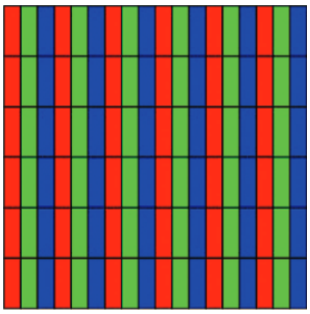


Slika 3.3: Pretvorba prevzorčene (2x) bitne slike (levo) v sivinsko (desno)

Slika 3.4: Glajen znak 'g', prikazan v različnih ločljivostih [13]

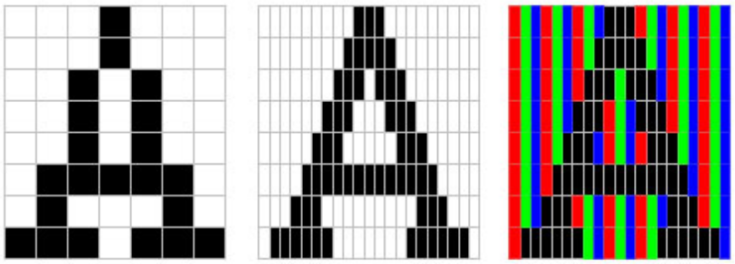
3.3 Podtočkovno glajenje teksta

Podtočkovno glajenje (ang. subpixel smoothing) lahko uporabimo le v primeru, ko uporabljamo slikovno prikazovalno napravo, na kateri lahko naslovimo podtočke (ang. subpixels) [18]. Ena izmed takšnih naprav je LCD zaslon, kjer je vsaka slikovna točka predstavljena s tremi podtočkami: rdečo (R), zeleno (G) in modro (B), kot to prikazuje Slika 3.5.



Slika 3.5: Organizacija podtočk na LCD zaslonu

Uporaba podtočk nam omogoča upodabljanje pisav s trikratno horizontalno ločljivostjo, kot to prikazuje Slika 3.6.



Slika 3.6: Od leve proti desni: bitna upodobitev znaka, podtočkovna upodobitev znaka, kot jo vidi človeško oko in dejanska podtočkovna upodobitev znaka na LCD zaslonu [18]

Podtočkovno glajenje deluje tako, da si od sosednjih slikovnih točk sposoja po eno ali dve podtočki [18], ter na tak način zgladi prej nazobčane robove. Po glajenju človeško oko ne bo zaznalo barvnih nepravilnosti, saj ne zmore zaznati vrstnega reda aktivnih podtočk, ki so vedno združene skupaj kot: RGB, BGR, ali BRG. Bralec lahko to preveri na Sliki 3.6. Pri tej metodi glajenja je vsak znak predstavljen kot RGB slika. Glajenje pa je primerno tudi za manjše velikosti pisav.

Poglavje 4

Pristopi k načrtovanju uporabniških vmesnikov za vgrajene sisteme

Pri načrtovanju uporabniških vmesnikov za vgrajene sisteme moramo upoštevati vsa pravila in dobre prakse, kot bi jih tudi sicer pri načrtovanju navadnih vmesnikov [1]. Paziti moramo na strukturo vmesnika, ki mora združevati povezane elemente skupaj ter ločevati nepovezane. Uporabnik mora v vsakem trenutku vedeti, v katerem sklopu vmesnika se nahaja, ter na kakšen način lahko med sklopi preklaplja. Paziti moramo na preprostost, saj moramo uporabniku našega vmesnika omogočiti, da lahko preproste in ponavljujoče naloge opravi hitro ter enostavno. Z uporabnikom je potrebno komunicirati v njegovem jeziku na jasen in enostaven način. Prikazati mu je potrebno vse informacije, ki jih potrebuje, da lahko opravi določeno nalogo, ob tem pa ga ne smemo obremeniti z nepotrebnimi podatki, ki jih v danem trenutku ne potrebuje. Obveščati ga moramo o napakah, ki so se zgodile kot posledica njegovih dejanj ali zaradi nedelovanja sistema. Nenazadnje pa moramo poskrbeti tudi za privlačnost uporabniškega vmesnika s samo postavitvijo elementov ter izbiro barvnih odtenkov. Uporabniški vmesnik bo postal okno v naš sitem, preko katerega si bo uporabnik ustvaril mnenje o naši napravi.

Posebnosti vgrajenih sistemov pa narekujejo nadaljne omejitve pri načrtovanju uporabniških vmesnikov. Omejitve so predvsem posledica izbire strojne ali programske opreme, s katero mora naš vmesnik delovati. Pri načrtovanju vgrajenega sistema je tako ključno, da kljub omejitvam obdržimo željeno funkcionalnost [8]. Slednje velja tudi za načrtovanje vgrajene programske opreme.

V nadaljevanju bomo podrobneje spoznali nekaj pomembnih pristopov k načrtovanju vgrajenih uporabniških vmesnikov. Spoznali bomo sledeče:

- uporabniško usmerjeno načrtovanje
- upoštevanje lastnosti in omejitev strojne opreme
- upoštevanje lastnosti in omejitev programske opreme

4.1 Uporabniško usmerjeno načrtovanje

Načrtovanje učinkovitega uporabniškega vmesnika za vgrajene sisteme se začne s spoznanjem, da je uporabniški vmesnik pomemben del vgrajenega sistema ter s postavitvijo uporabnika v središče našega načrtovanja in razvojnega procesa [6]. Takšnemu pristopu rečemo uporabniško usmerjeno načrtovanje (ang. user-centered design), pri katerem se moramo že zelo zgodaj osredotočiti na uporabnika in njegove naloge. Temelji na iterativnem načrtovanju s spiralnim modelom, ki predpisuje več iteracij načrtovanja, implementiranja in vrednotenja. Zgodnje iteracije so cenejše in prilagodljivejše, saj z uporabo prototipov (papirnatih in računalniških) že zelo zgodaj odkrijemo pomankljivosti. Posledično nam ni potrebno zavreči že napisane izvorne kode. V vsaki iteraciji moramo vrednotiti narejeno oziroma načrtano, vrednotiti pa morajo tako strokovnjaki kot končni uporabniki. Za hitrejši razvoj je priporočljivo uporabljati že obstoječa integrirana razvojna ogrodja (ang. IDE). Po končni implementaciji pa je potrebno izvesti testiranje uporabnikov in spremljati njihove odzive ob uporabi vmesnika [9].

4.2 Načrtovanje z upoštevanjem omejitev strojne opreme

Vgrajeni sistemi so si lahko medsebojno zelo različni glede na lastnosti in zmogljivost strojne opreme. Pravilno je, da se zavedamo teh omejitev ter zavedanje o njih vključimo v načrtovanje uporabniškega vmesnika. Uporabnik se jih posledično ne bo zavedal, njegova uporabniška izkušnja pa bo ostala pozitivna.

Poglejmo si nekaj možnih omejitev strojne opreme:

- zmogljivost centralno procesne enote
- količina delovnega pomnilnika
- velikost zaslona
- funkcionalno omejene vhodne naprave

4.2.1 Zmogljivost centralno procesne enote

Zmogljivost centralno procesne enote (CPE) vpliva na količino in zahtevnost grafičnih gradnikov uporabniškega vmesnika, ki jih lahko prikažemo uporabniku v danem trenutku. Pogosto je CPE edina enota v sistemu, ki jo lahko uporabimo za izračun in obdelavo grafičnih podatkov. Izjema so modernejši sistemi na čipu (ang. SoC), ki že vsebujejo grafične pospeševalnike. Odločiti se moramo tudi, koliko procesne moči bomo namenili grafičnemu uporabniškemu vmesniku, saj mora vgrajena naprava pogosto opravljati tudi druge računsko zahtevne operacije, ki so pogosto prvotnega pomena.

4.2.2 Količina delovnega pomnilnika

Grafični uporabniški vmesniki pogosto uporabljajo vnaprej pripravljene grafične podobe, kot so slike elementov grafičnega vmesnika. To so lahko npr. slike gumbov, drsnikov, vnosnih polj, slike ozadja ter celo vnaprej pripravljene

animacije. Količina delovnega pomnilnika vpliva na število slikovnih podatkov, ki jih lahko hranimo v njem. Predstavljajmo si napravo, na kateri imamo na voljo le nekaj kilobajtov delovnega pomnilnika. Za njo bi morali zasnovati bistveno enostavnejši grafični vmesnik, kot bi ga lahko za napravo, kjer te omejitve ni. Pogost pristop pri zmanjševanju potrebe po delovnem pomnilniku je možnost izklopa prevajanja dela programske kode, ki jo na danem vgrajenem sistemu ne potrebujemo. V primeru uporabe statičnega prevajanja pa lahko takšno optimizacijo namesto nas opravi že prevajalnik programske kode (npr. GCC). Predstavljajmo si, da imamo na voljo zelo obsežno programsko knjižnico za izris gradnikov uporabniškega vmesnika, mi pa potrebujemo le del njene funkcionalnosti. Programska koda, ki se ne bo uporabljala, bo le po nepotrebnem zasedala pomnilnik.

4.2.3 Velikost prikazovalne naprave (zaslona)

Velikost prikazovalne naprave oziroma zaslona pomembno vpliva na število in velikost grafičnih elementov, ki jih lahko v danem trenutku prikažemo uporabniku. Posledično to vpliva tudi na strukturo uporabniškega vmesnika, saj moramo na manjših zaslonih prikazati informacije na bistveno bolj kompakten način, hkrati pa ne smemo izgubiti uporabnosti ter preglednosti. Slika 4.1 prikazuje uporabniški vmesnik na majhnem LCD zaslonu.

Slika 4.1: Primer uporabniškega vmesnika na štiri vrstičnem LCD zaslonu

4.2.4 Zasloni občutljivi na dotik

Zasloni, občutljivi na dotik, pomembno vplivajo na zasnovo uporabniškega vmesnika, saj pogojujejo zasnovo gradnikov uporabniškega vmesnika. Sle-

dnji morajo biti dovolj veliki, razmik med njimi pa zadosten, da jih lahko uporabnik brez težav pritisne oziroma izbere. V nasprotnem primeru bo imel uporabnik veliko težav pri upravljanju uporabniškega vmesnika. V kolikor ne želimo pokvariti izgleda vmesnika s prevelikimi gradniki, lahko to rešimo na dva načina. Pri prvem narišemo gradnik v željeni velikosti, področje, kjer bo aktiven, pa mu določimo tako, da bo večje od njegove slikovne predstavitve (ang. iceberg tips). Pri drugem pa algoritmično ugotavljamo naslednji gradnik, ki ga bo uporabnik pritisnil in ob tem povečamo aktivno področje okoli njega (ang. adaptive targets) [5]. Slednji pristop je zelo uporaben pri navideznih tipkovnicah (ang. on-screen keyboards).

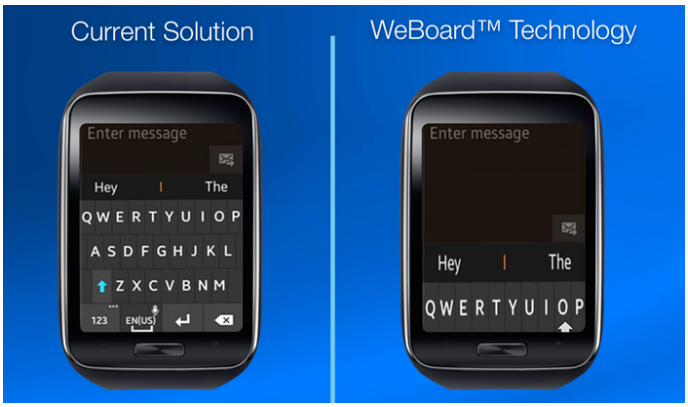
4.2.5 Barvni model slikovnega prikazovalnika

Barve vnesejo v uporabniški vmesnik dodatno preglednost ter povečajo oziroma zmanjšajo težo prikazanih informacij. Primer: opozorila lahko prikažemo v rdeči barvi, informacije o pravilnem delovanju pa v zeleni.

4.2.6 Funkcionalno omejene vhodne naprave

Vhodne naprave, namenjene upravljanju uporabniških vmesnikov, so na vgrajenih sistemih pogosto funkcionalno omejene. Snovalci vgrajenih sistemov imajo zato mnogokrat prostorske, stroškovne ali kakšne druge omejitve, ki se posledično odražajo v izbiri vhodnih naprav. Uporabniški vmesnik moramo posledično zasnovati na način, da uporabniku omogočimo lahkotno ter učinkovito navigacijo preko vseh delov našega vmesnika ne glede na omejitve. Bralec lahko sam premisli, kako bi zasnoval uporabniški vmesnik, kadar bi imel na voljo zaslon, občutljiv na dotik, ter kako, ko bi imel na voljo le preprost krožni kodirnik. Kako v slednjem primeru izpolniti vhodna tekstovna polja? Lahko npr. oblikujemo navidezno tipkovnico, kot jo prikazuje desni del Slike 4.2 (WeBoard™ Technology). Ta je primerna za uporabo na majhnih slikovnih prikazovalnikih, upravljati pa jo je možno tudi s krožnim kodirnikom. Črke lahko izbiramo z vrtenjem gumba levo ali desno, izberemo

pa jo s pritiskom na gumb.



Slika 4.2: Apple Watch [24] tipkovnici: standardna (levo) ter WeBoard™ Technology (desno)

4.3 Načrtovanje z upoštevanjem omejitev programske opreme

V nadaljevanju si bomo pogledali, kako načrtovati uporabniški vmesnik v primeru odsotnosti operacijskega sistema. Nato pa bomo spoznali še nekaj kriterijev za izbiro programskega jezika oziroma prevajalnika.

4.3.1 Odsotnost operacijskega sistema

Številni vgrajeni sistemi niso zasnovani na operacijskem sistemu oziroma le tega nimajo. V tem primeru smo prikrajšani za vse storitve, ki jih ta ponuja. V kolikor želimo, da bi naš uporabniški vmesnik deloval tudi na njih, moramo mankajočo funkcionalnost dodati sami ali pa zasnovati naš uporabniški vmesnik tako, da teh ne bo potreboval.

Poglejmo si nekaj storitev operacijskega sistema, ki jih tipično uporabljamo v uporabniških vmesnikih:

- upravljanje s pomnilnikom oziroma zaseg pomnilnika
- dostop do izhodnih prikazovalnih naprav
- prejemanje vhodnih dogodkov z vhodnih naprav

Upravljanju s pomnilnikom se lahko izognemo tako, da uporabljamo statične spremenljivke. Primer: namesto da bi dinamično zasegli 32 bajtov pomnilnika za hranjene niza, lahko že vnaprej predvidimo npr. 256 bajtov statičnega prostora. Lahko pa napišemo lastno implementacijo za upravljanje s pomnilnikom. Potrebovali bi vsaj funkciji za zaseg pomnilnika (ang. malloc) ter za njegovo sprostitev (ang. free). Lahko bi npr. uporabili algoritem Buddy Systems [10].

Uporabi vhodno-izhodnih naprav pa se žal ne moremo izogniti. Posledično bomo morali programsko kodo za dostop napisati sami. Neposreden dostop ni priporočljiv, saj bomo tako uporabniški vmesnik vezali na točno določen tip naprav. Bolje je do njih dostopati posredno, preko splošnonamenskih vmesnikov, saj bomo lahko tako na enostaven način razširili nabor podprtih vhodno-izhodnih naprav, ko bo to potrebno. Obstoječe rešitve za izdelavo vgrajenih uporabniških vmesnikov imajo tipično zelo širok nabor podprtih vhodno-izhodnih naprav, predvsem gonilnike za različne krmilnike slikovnih prikazovalnikov (SSD1306, SEPS525 ...).

4.3.2 Izbira programskega jezika

Pred izdelavo programske opreme za vgrajene sisteme je potrebno izbrati ustrezen programski jezik. Na našo odločitev lahko vplivajo številni kriteriji:

- prevajalniki, ki so nam na voljo za dano arhitekturo
- velikost prevedene kode
- hitrost prevedene kode
- potreba po objektnem programiranju
- potreba po integraciji naše kode v že obstoječe programske rešitve

Poglavje 5

Načrt razvoja grafičnega uporabniškega vmesnika za naprave Xiphra SE

Pred izdelavo uporabniškega vmesnika smo naredili načrt, v katerem smo že v prvem koraku naredili analizo uporabnikov in njihovih nalog. Poskušali smo ugotoviti način, na katerega bodo uporabniki uporabljali naše naprave in slednje upoštevati pri nadaljnem načrtovanju. Osredotočili smo se predvsem na ponavljajoče naloge, ki jih bodo uporabniki pogosto opravljali. Slednje smo poskušali razumeti ter poenostaviti.

Nato smo opredelili strojno in programsko opremo ter izbrali programski jezik. Pri načrtovanju smo se naslonili na vsebino prejšnjega poglavja, saj smo se morali zavedati vseh omejitev strojne in programske opreme. Načrtovali smo s pomočjo papirnatih prototipov (računalniških skic uporabniškega vmesnika), s katerimi smo lahko hitro vrednotili naše ideje oziroma rešitve.

Uporabniško usmerjeno načrtovanje priporoča uporabo že obstoječih ogrodij in programskih knjižnic za izdelavo uporabniških vmesnikov. Posledično smo morali pred izdelavo uporabniškega vmesnika narediti pregled obstoječih rešitev in se odločiti za najustreznejšo glede na naše zahteve. Po skrbnem pre-

gledu obstoječih rešitev smo sprejeli to odločitev, ki je podrobneje opisana in utemeljena v podpoglavju 5.6. Pri odločitvi smo upoštevali predhodno izdelan načrt, v katerem smo strnili vse naše potrebe ter željene lastnosti končne rešitve.

5.1 Analiza uporabnikov in njihovih nalog

Naprave Xiphra so namenjene šifriranju mrežnega prometa. Uporabnik bo moral pred uporabo naše naprave nastaviti vse potrebne omrežne nastavitve, da se bo lahko naprava povezala z ostalimi šifrirnimi napravami ter med njimi ustvarila varne prehode oziroma tunele. Ker je število različnih sklopov omrežnih nastavitev veliko, bomo morali uporabnikom omogočiti bližnjice s katerimi jim bomo pohitrili ponavljajoče operacije (vnos IP naslovov, mask, prehodov ...). Že en sam napačen vnos lahko povzroči nepravilno delovanje naprave, med mnogimi sklopi pa je takšen vnos težko najti in popraviti. Posledično moramo uporabnikom naših naprav onemogočiti nepravilno izpolnjevanje vhodnih polj. Po pravilni nastavitvi vseh omrežnih sklopov postane šifrirna naprava samostojna, uporabnik pa bo na njej le še občasno spremljal informacije o njenem delovanju (temperatura, verzija programske opreme, tip licence ...). Uporabniki naših naprav stremijo k enostavnosti in učinkovitosti uporabniškega vmesnika, preko katerega bodo lahko v čim krajšem času aktivirali vse svoje šifrirne naprave.

5.2 Opredelitev strojne opreme

Uporabniški vmesnik bo moral delovati na Xiphra SE šifrirnih napravah, ki vsebujejo sledečo strojno opremo:

- 800 Mhz CPE (ARMv7)
- 128 MB delovnega pomnilnika
- 4 GB flash pomnilnika (eMMC)

- OLED zaslon, ločljivost: 160x128, barvni model: RGB565, RGB666
- krožni kodirnik kot edino vhodno napravo

Kot lahko vidimo, se omejitve pojavljajo predvsem zaradi majhne velikosti slikovnega prikazovalnika ter tudi zaradi omejene vhodne naprave. Pri količini delovnega pomnilnika se bomo dodatno omejili le na 4 MB, saj želimo preostanek prepustiti drugim uporabniškim procesom, ki so za delovanje sistema prvotnega pomena. Zaradi majhne velikosti slikovnega prikazovalnika bodo morali biti gradniki uporabniškega vmesnika enostavni, pisave pa primerno majhne, berljive ter podtočkovno glajene. Navigacija prek gradnikov bo morala biti kljub omejeni vhodni napravi enostavna in učinkovita. Potrebovali pa bomo tudi navidezno tipkovnico, preko katere bomo izpolnjevali vhodna polja.

Potrebe, ki izhajajo iz opredelitve strojne opreme:

- majhna poraba delovnega pomnilnika (4 MB)
- enostavni gradniki uporabniškega vmesnika
- navidezna tipkovnica (gradnik uporabniškega vmesnika)
- podtočkovno glajenje teksta
- podpora za krožni kodirnik

5.3 Opredelitev programske opreme

Uporabniški vmesnik bo moral delovati tako v sistemu Linux kot tudi na sistemih brez operacijskega sistema. Pravzaprav si želimo in moramo zasnovati uporabniški vmesnik na način, ki bo omogočal enostaven prehod na katerikoli željen sistem. To lahko dosežemo z uporabo splošnonamenskih vhodno-izhodnih vmesnikov. Prav tako se želimo izogniti dinamičnemu zasegu pomnilnika, saj ta pogosto ni na voljo na sistemih brez operacijskega sistema.

Potrebe, ki izhajajo iz opredelitve operacijskega sistema:

- delovanje up. vmesnika v sistemu Linux
- delovanje up. vmesnika na sistemih brez operacijskega sistema
- statično zaseganje pomnilnika
- uporaba slikovnega medpomnilnika v sistemu Linux

5.4 Izbira programskega jezika

Na voljo imamo tako prevajalnik za objektni programski jezik C++ kot tudi za programski jezik C. Odločitev med njima je težka, saj nam objektni način programiranja pride zelo prav še posebno zaradi dejstva, da so uporabniški vmesniki tipično objektno zasnovani. Za programski jezik C smo se odločili zaradi potrebe po integraciji z že obstoječo programsko opremo "u-boot", ki je napisana v tem jeziku. V kolikor bomo uporabili katero izmed obstoječih rešitev za izdelavo uporabniškega vmesnika, bo ta morala biti napisana v enakem jeziku. Izvorna koda rešitve pa dostopna ter kakovostna.

Potrebe, ki izhajajo iz izbire programskega jezika:

- programski jezik C
- dosegljivost programske kode
- kakovost programske kode

5.5 Prototip uporabniškega vmesnika

Namen imamo izdelati uporabniški vmesnik, preko katerega lahko spreminjamo nastavitve Xiphra SE šifrirne naprave. Zasnovali ga bomo preprosto in učinkovito, saj želimo uporabniku naše naprave omogočiti enostaven dostop do vseh podsklopov našega vmesnika. Posledično bomo omejili globino uporabniškega vmesnika na največ tri nivoje. Posebnost našega uporabniškega vmesnika bo tudi navidezna tipkovnica, prilagojena majhnemu

slikovnemu prikazovalniku ter ponavljajočemu vnašanju zelo podobnih nastavitev. Načrtovan izgled navidezne tipkovnice prikazuje Slika 5.3, ki se nahaja na koncu tega podpoglavja. Tipkovnica je izrisana nad prvim vnosnim poljem.

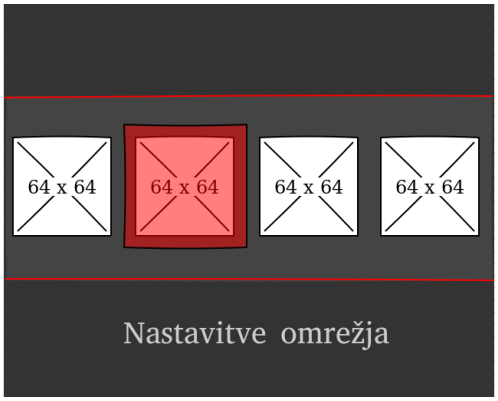
Zahtevane lastnosti navidezne tipkovnice:

- po velikosti mora ustrezati slikovnemu prikazovalniku
- vnašanje in brisanje znakov
- potrditev in prekinitev vnosa
- bližnjice za ponavljujoče operacije (kopiraj, prilepi ...)
- onemogočati mora napačne vnose

Z uporabo programa Pencil [21] smo si skicirali glavne sklope in podsklope uporabniškega vmesnika. Tako smo veliko lažje prenesli vse naše oblikovne ali drugačne zamisli v končno rešitev. Sklope in podsklope smo zasnovali enostavno in pregledno. Barvne odtenke pa smo uskladili s celostno podobo vgrajene naprave. Skice predstavljajo prototip uporabniškega vmesnika, s katerim smo poskušali zaznati pomankljivosti že v zgodnjih fazah razvoja. Vrednotili smo ga sami, saj nam narava dela z našimi strankami ni omogočala zgodnjega uporabniškega vrednotenja, to bo opravljeno šele v fazi računalniškega prototipa. Skice so nam služile tudi kot seznam vseh potrebnih gradnikov uporabniškega vmesnika.

Slika 5.1 prikazuje skico glavnega izbirnega menija, preko katerega bomo izbirali glavne podsklope našega vmesnika. Zasnovali smo ga s pomočjo ikon, med katerimi se bo uporabnik sprehajal z uporabo krožnega kodirnika. Ob potrditvi izbire bomo skočili ali na dodatni podmeni, kot ga prikazuje Slika 5.2, ali pa neposredno na enega izmed podsklopov za nastavljanje. Podsklop za nastavljanje omrežnih nastavitev prikazuje Slika 5.3.

5.5.1 Glavni meni

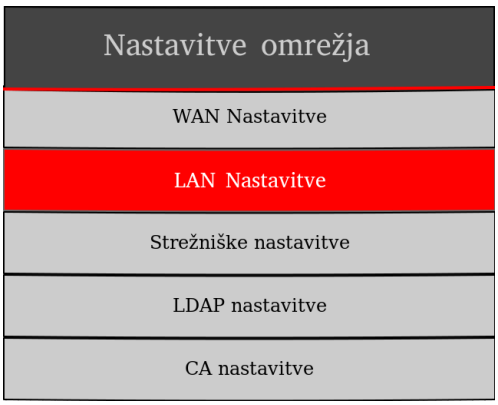


Slika 5.1: Skica glavnega izbirnega menija

Potrebni gradniki uporabniškega vmesnika:

- slika (ang. image)
- tekstovno polje (ang. text field)
- slikovne operacije: črta, pravokotnik

5.5.2 Izbirni podmeni

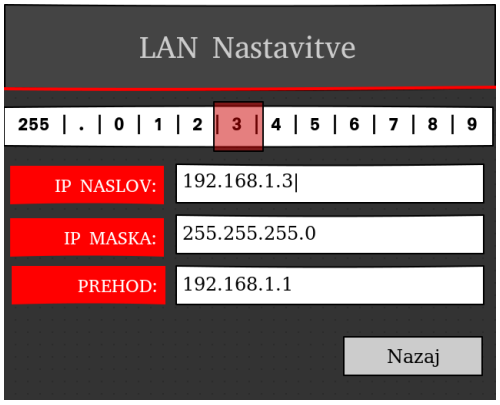


Slika 5.2: Skica izbirnega podmenija

Potrebni gradniki uporabniškega vmesnika:

- gumb
- tekstovno polje
- slikovne operacije: črta, pravokotnik

5.5.3 Podsklop za nastavljanje omrežnih nastavitev



Slika 5.3: Skica podsklopa za nastavljanje mrežnih nastavitev in prikazom navidezne tipkovnice

Potrebni gradniki uporabniškega vmesnika:

- tekstovno polje
- vnosno polje
- navidezna tipkovnica
- gumb
- slikovne operacije: črta, pravokotnik

5.6 Pregled obstoječih rešitev za izdelavo uporabniških vmesnikov

Pred implementacijo uporabniškega vmesnika smo naredili analizo primernosti nekaterih obstoječih integriranih razvojnih ogrodij in programskih knjižnic za izdelavo vgrajenih uporabniških vmesnikov.

5.6.1 Qt Embedded

Qt je programsko ogrodje namenjeno izdelavi uporabniških vmesnikov. Napisano je v programskem jeziku C++ ter deluje na številnih sistemih: Linux, OS X, Windows, QNX, Android, iOS, BlackBerry, Sailfish OS in še mnogo drugih. Razvoj ogrodja Qt se je začel že v letu 1990 in se vse do danes neprestano razvija. Prvotna razvijalca ogrodja Qt sta bila norveška programerja Eirik Chambe-Eng in Haavard Nord, sedaj pa ga ima v lasti podjetje The Qt Company. Razširjeno ogrodje Qt Embedded pa ponuja še nekaj dodatnih možnosti, kot so izklop prevajanja delov programske kode ogrodja, uporabo različnih izrisovalnih vmesnikov, uporabo različnih vhodnih vmesnikov ter podporo za številne priljubljene vgrajene sisteme. Primer uporabniškega vmesnika, izdelanega z uporabo ogrodja Qt Embedded prikazuje Slika 5.4.

Po skrbnem pregledu ogrodja smo ugotovili, da rešitev ne ustreza vsem našim zahtevam iz prejšnjega poglavja. Gradniki uporabniškega vmesnika niso zasnovani enostavno in so posledično preveliki za prikaz na manjših zaslonih. Ogrodje je napisano v programskem jeziku C++, kar bi zelo otežilo integracijo z obstoječo programsko opremo. Podporo za krožni kodirnik bi morali dodati sami. Uporabniški vmesnik pa ne bi deloval brez operacijskega sistema ter bi porabil preveč pomnilnika. Pregled je strnjen v Tabeli 5.1, v kateri so z rdečo barvo označene nezaželene lastnosti te rešitve.

Slika 5.4: Primer Qt Embedded uporabniškega vmesnika [15]

Qt Embedded	
programski jezik:	**C++**
operacijski sistem:	Linux, Windows, Android, iOS, ...
samostojno delovanje:	**NE, potrebuje OS**
poraba delovnega pomnilnika:	**> 4MB**
statično zaseganje pomnilnika:	**NE, samo dinamično**
slikovni medpomnilnik:	DA
podprte zaslonske ločljivosti:	ni posebnih omejitev
barvni formati:	RGB565, RGB666, RGB888, sivinski, monokromatski, ...
vsi gradniki up. vmesnika:	DA
navidezna tipkovnica:	**NE**
enostavni gradniki:	**NE**
profesionalen izgled gradnikov:	DA
navadno glajenje teksta:	DA

podtočkovno glajenje teksta:	DA
podpora za krožni kodirnik:	**NE, možno dodati**
dostopna izvorna koda:	DA
kvalitetna izvorna koda:	DA
strošek nakupa:	**podatek ni na voljo**

Tabela 5.1: Pregled lastnosti ogrodja Qt Embedded

5.6.2 easyGUI

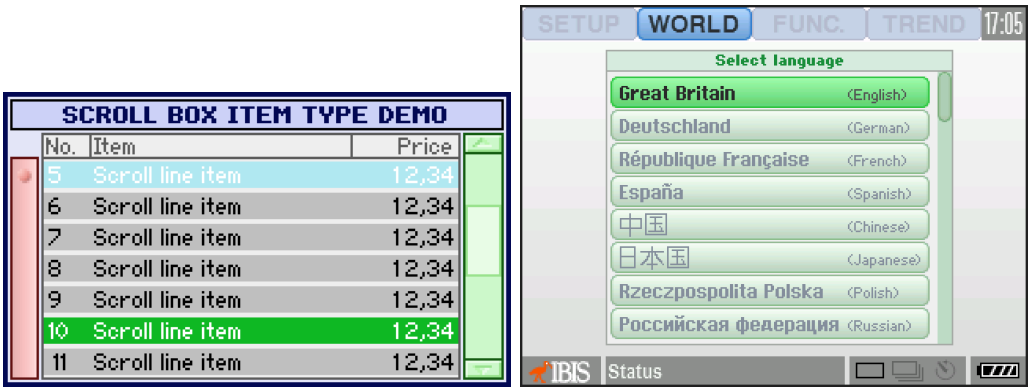
EasyGUI je programsko ogrodje, namenjeno izdelavi uporabniških vmesnikov za izključno vgrajene sisteme. Napisano je v programskem jeziku C ter deluje na številnih sistemih, med drugimi tudi v sistemu Linux in sistemih brez operacijskega sistema. Podjetje IBIS Solutions Aps ga razvija že od leta 1990 in ga vse do danes neprestano izboljšuje. Naredili so ga zaradi lastnih potreb ter zaradi dejstva, da v tistem času niso našli primerne obstoječe rešitve. Komercialno dostopen je od leta 2004.

Dejstvo, da je ogrodje zasnovano izključno za vgrajene sisteme, ponuja številne prednosti:

- optimizacija velikosti prevedene kode
- optimizacija porabe delovnega pomnilnika
- podpora za številne vgrajene sisteme
- podpora za številne LED prikazovalnike (gonilniki)
- podpora za številne barvne formate
- vsebuje številne fonte, ki so primerni za majhne zaslonske ločljivosti

Navkljub vsem pozitivnim lastnostim ogrodja smo po skrbnem pregledu ugotovili, da ne ustreza vsem našim zahtevam. Z nakupom te rešitve ne

bi pridobili izvorne kode. Posledično bi morali plačevati letne premije za vzdrževanje in dodajanje manjkajoče funkcionalnosti. Odsotna je podpora za krožni kodirnik in navidezno tipkovnico. Sam izgled gradnikov uporabniškega vmesnika pa se nam ni zdel dovolj dober, da bi upravičil razmeroma velik strošek nakupa. Ocena je seveda subjektivna in se bralec z njo mogoče ne bi strinjal. Primere uporabniških vmesnikov, narejenih z ogrodjem easyGUI prikazuje Slika 5.5. Pregled pa smo strnili v Tabeli 5.2, v kateri so z rdečo barvo označene nezaželene lastnosti te rešitve.



Slika 5.5: Primer uporabniških vmesnikov narejenih z ogrodjem easyGUI [17]

easyGUI	
programski jezik:	C
operacijski sistem:	Linux ter mnogi drugi
samostojno delovanje:	DA, OS ni potreben
poraba delovnega pomnilnika:	$<$ 4MB
statično zaseganje pomnilnika:	**podatek ni na voljo**
slikovni medpomnilnik:	DA
podprte zaslonske ločljivosti:	ni posebnih omejitev

barvni formati:	RGB565, RGB666, RGB888, sivinski, monokromatski, ...
vsi gradniki up. vmesnika:	DA
navidezna tipkovnica:	**NE**
enostavni gradniki:	DA
profesionalen izgled gradnikov:	**NE**
navadno glajenje teksta:	DA
podtočkovno glajenje teksta:	**NE**
podpora za krožni kodirnik:	**NE**
dostopna izvorna koda:	**NE**
kvalitetna izvorna koda:	/
strošek nakupa:	**1019 - 2109 EUR**

Tabela 5.2: Pregled lastnosti ogrodja easyGUI

5.6.3 μGFX

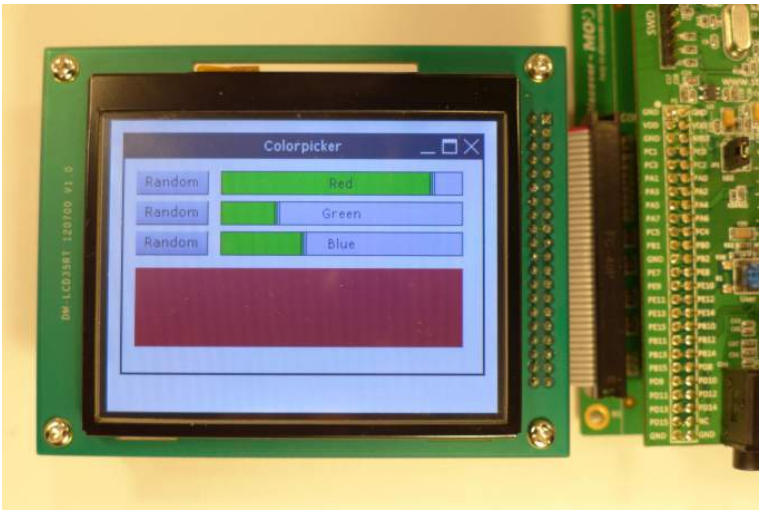
μGFX je programska knjižnica, napisana v programskem jeziku C ter je namenjena izdelavi uporabniških vmesnikov za izključno vgrajene sisteme [27]. Razvilo jo je švicarsko podjetje B-Electronics. Od leta 2012 dalje pa je projekt v solasti avstralskega podjetja InMarket Systems Pty Ltd. Od ostalih pregledanih rešitev se razlikuje po tem, da vsebuje podporo za več zaslonov hkrati, podporo za oddaljeno prikazovanje slike (ang. remote displays), obenem pa je zmožna predvajati zvočne posnetke. Uporabimo jo lahko za izdelavo uporabniškega vmesnika tako za sisteme z operacijskim sistemom kot za tiste brez njega.

Dejstvo, da je knjižnica zasnovana izključno za vgrajene sisteme, ponuja številne prednosti:

- optimizacija velikosti prevedene kode (izklop neuporabljenih modulov)
- majhna poraba delovnega pomnilnika
- podpora za številne vgrajene sisteme
- podpora za številne LED prikazovalnike (gonilniki)
- podpora za številne barvne formate
- možna uporaba lastnih fontov (preko programa za pretvorbo)

μGFX je edina obstoječa rešitev izmed treh, ki neposredno podpira krožni kodirnik, hkrati pa ima tudi najmanj pomankljivosti. Slednje je vodilo v resen razmislek o nakupu. Zaradi javno dostopne izvorne kode smo sprejeli odločitev, da pred nakupom preverimo njeno kakovost, saj bi lahko slaba programska koda povzročila nemalo težav. Ob pregledu smo ugotovili, da je koda zelo slabo napisana in ne upošteva dobrih programerskih praks. Ob dejstvu, da knjižnica uporablja dinamično zaseganje pomnilnika, smo sprejeli odločitev, da te rešitve ni varno uporabiti. Bralec lahko sam preveri, če se z napisanim strinja [28]. Primer uporabniškega vmesnika, narejenega s

knjižnico μGFX prikazuje Slika 5.6, pregled pa smo strnili v Tabeli 5.3, v kateri so z rdečo barvo označene nezaželene lastnosti te rešitve.



Slika 5.6: Primer uporabniškega vmesnika narejenega s knjižnico μGFX [29]

μGFX	
programski jezik:	C
operacijski sistem:	Linux, Arduino, Free RTOS, ...
samostojno delovanje:	DA, OS ni potreben
poraba delovnega pomnilnika:	$<$ 4MB
statično zaseganje pomnilnika:	**NE, samo dinamično**
slikovni medpomnilnik:	DA
podprte zaslonske ločljivosti:	ni posebnih omejitev
barvni formati:	RGB565, RGB666, RGB888, sivinski, monokromatski, ...
vsi gradniki up. vmesnika:	DA

navidezna tipkovnica:	**DA, toda neustrezna**
enostavni gradniki:	DA
profesionalen izgled gradnikov:	DA
navadno glajenje teksta:	DA
podtočkovno glajenje teksta:	**NE**
podpora za krožni kodirnik:	DA
dostopna izvorna koda:	DA
kvalitetna izvorna koda:	**NE**
strošek nakupa:	100 - 1650 EUR

Tabela 5.3: Pregled lastnosti knjižnice μGFX

5.6.4 Primerjava in sklep

Obstoječe rešitve smo primerjali s pomočjo Tabele 5.4, v kateri smo označili podprte lastnosti. Opazimo lahko, da nobena izmed rešitev ni v popolnosti ustrezala našim zahtevam. Posledično smo se odločili izdelati lastne podporne knjižnice, saj nam vnaprej zastavljenem časovnem okviru ni uspelo najti ustrezne obstoječe rešitve. Pomankljivosti so se pokazale predvsem v odsotnosti podpore za krožni kodirnik in navidezno tipkovnico, kompleksnosti gradnikov, preveliki porabi delovnega spomina, dinamičnemu zaseganju pomnilnika ter v nekvalitetni programski kodi. Naše podporne knjižnice bodo morale odpraviti vse te pomankljivosti, izdelane pa bodo morale biti v razumnih stroškovnih okvirih (max. 100 delovnih ur).

	Qt Embedded	**easyGUI**	**μGFX**
programski jezik C		x	x
sistem Linux	x	x	x
samostojno delovanje		x	x
poraba pomnilnika		x	x
statično zaseganje		/	
slikovni medpomnilnik	x	x	x
zaslonske ločljivosti	x	x	x
barvni formati	x	x	x
vsi gradniki up. vmesnika	x	x	x
navidezna tipkovnica			
enostavni gradniki		x	x
izgled gradnikov	x		x
glajenje teksta	x	x	x
podtočkovno glajenje	x		
podprt krožni kodirnik			x
dostopna izvorna koda	x		x
kvalitetna izvorna koda	x	/	
strošek nakupa	x		x

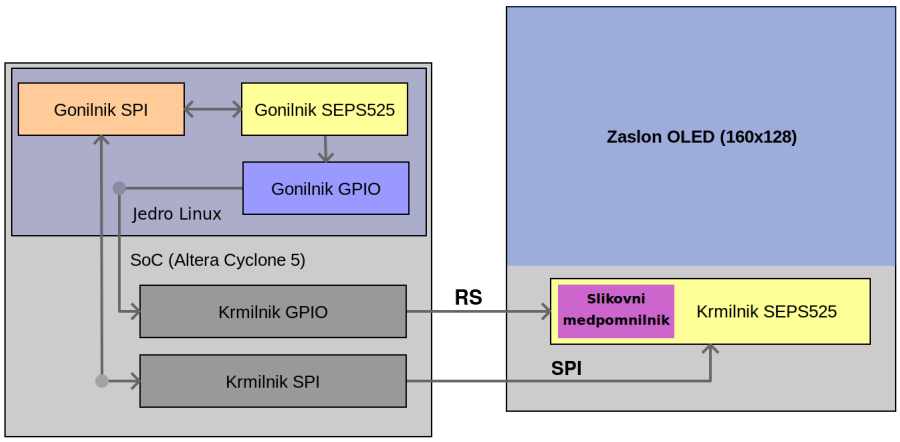
Tabela 5.4: Primerjava obstoječih rešitev za izdelavo vgrajenih uporabniških vmesnikov

Poglavje 6

Razvoj gonilnika slikovnega medpomnilnika (SEPS525)

Xiphra SE šifrirne naprave imajo vgrajen slikovni prikazovalnik, ki je osnovan na krmilniku SEPS525. Povezan je na Cyclone V SoC preko podatkovnega vodila SPI, kot to prikazuje Slika 6.1. Do krmilnika SEPS525 lahko dostopamo preko krmilnika SPI, ki ga nadzoruje in upravlja gonilnik SPI.

Po neuspešnem iskanju obstoječega gonilnika SEPS525 za jedro Linux, smo bili primorani napisati svojega, saj brez njega ne bi mogli dostopati do slikovnega medpomnilnika.



Slika 6.1: Slikovni prikazovalnik povezan preko podatkovnega vodila SPI

Pred izdelavo gonilnika slikovnega medpomnilnika za krmilnik SEPS525, povezanega preko podatkovnega vodila SPI, moramo zelo dobro poznati in razumeti sledeče:

- krmilnik SEPS525
- uporabo podatkovnega vodila SPI znotraj jedra Linux
- upravljanje GPIO nožic znotraj jedra Linux
- gonilniški vmesnik slikovnega medpomnilnika v jedru Linux
- osnove pisanja gonilnikov za jedro Linux

Napisati moramo namreč funkcije za dostop do registrov krmilnika SEPS525, v katerih je potrebno uporabiti obstoječe funkcije jedra Linux za prenos podatkov preko protokola SPI ter funkcije za upravljanje GPIO nožic. Krmilnik moramo nato, s pisanjem v registre, nastaviti v pravilno oziroma željeno stanje. Da napišemo gonilnik slikovnega medpomnilnika, moramo napisati ustrezne funkcije in izpolniti ustrezne podatkovne strukture ter slednje, znotraj funkcije "probe()", prijaviti jedru Linux. Potrebne funkcije in podatkovne strukture določa gonilniški vmesnik slikovnega medpomnilnika. Nato samo še prijavimo gonilnik jedru Linux.

6.1 Opis krmilnika SEPS525

Krmilnik SEPS525 vsebuje številne registre s katerimi lahko spreminjamo njegovo delovanje [23]. Do njih dostopamo preko podatkovnega vodila SPI in s pomočjo nožice RS, ki je povezana na krmilnik GPIO. Slednjo moramo, pred vsakim prenosom po vodilu SPI, ustrezno nastaviti. Visoka (1) vrednost pomeni, da bomo prek vodila SPI prenašali podatke, nizka (0) pa pomeni, da bomo prenašali ukaze. Posledično je vsak dostop do krmilnika SEPS525 sestavljen iz dveh delov. Najprej prenesemo ukaz (številko registra) in nato še podatke. Ukazi so vedno 8-bitni, medtem ko so lahko podatki različnih dolžin (8-bitni, 16-bitni in 18-bitni).

6.1.1 Registri krmilnika SEPS525

Prek registrov lahko nastavljamo barvne modele RGB, barvne modele BGR, obračamo in zamikamo prikazano sliko, spreminjamo izrisovalno okno, omogočimo in onemogočimo osveževalnik zaslona, nastavljamo hitrost osveževanja, ugasnemo oziroma prižgemo zaslon ter še mnogo več. V nadaljevanju si bomo pogledali nekaj najpomembnejših registrov, opisali pa bomo samo bitna polja, ki se nam zdijo pomembna. Bralca spodbujamo, da si ogleda priročnik za krmilnik SEPS525 [23].

REDUCE_CURRENT (04h)

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
R/W	-	-	-	-	-	RC	OSCPS	PS
privzeto:	0	0	0	0	0	0	0	0

- **PS** - Način ohranjevanja energije. SRN=0; normalno delovanje. SRN=1; zaslon je ugasnjen, analogni del pa v resetiranem stanju.

SOFT_RST (05h)

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
R/W	-	-	-	-	-	-	-	SRN
privzeto:	0	0	0	0	0	0	0	0

- **SRN** - Programska ponastavitev krmilnika. SRN=0; normalno delovanje. SRN=1; vrednosti vseh registrov se ponastavijo na privzete vrednosti.

DISP_ON_OFF (06h)

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
R/W	PREM	-	-	-	-	-	-	DON
privzeto:	0	0	0	0	0	0	0	0

- **DON** - Ugašanje in prižiganje zaslona. DON=0; zaslon je ugasnjen. DON=1; zaslon je prižgan.

DISPLAY_MODE_SET (13h)

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
R/W	SWAP	SM	RD	CD	-	SPT	DC1	DC0
privzeto:	0	0	0	0	0	0	0	0

- **SWAP** - Pretvorba barvnega formata RGB v BGR. SWAP=1; slikovna točka sprejeta v barvnem formatu RGB se pretvori v barvni model BGR. SWAP=0; slikovna točka se ne pretvori.

RGB_IF (14h)

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
R/W	-	-	RIM1	RIM0	-	-	-	EIM
privzeto:	0	0	0	1	0	0	0	1

- **EIM** - tip zunanjega vmesnika. EIM=0; zunanji vmesnik RGB. EIM=1; zunanji vmesnik MPU. V našem primeru moramo nastaviti zunanji vmesnik MPU.

MEMORY_WRITE_MODE (16h)

Krmilnik SEPS525 vsebuje horizontalni in vertikalni naslovni števec. Oba uporablja pri vpisovanju sprejetih slikovnih točk v slikovni medpomnilnik. Po vsakem vpisu poveča oziroma zmanjša naslovna števca glede na nastavitve v tem registru. Nastavitve tako vplivajo na smer izrisovanja po horizontalni oziroma vertikalni smeri.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
R/W	-	DFM1	DFM0	TRI	-	HC	VC	HV
privzeto:	0	0	0	0	0	1	1	0

- **HC** - horizontalno povečevanje oziroma zmanjševanje naslova. HC=0; horizontalni naslovni števec se zmanjša. HC=1; horizontalni naslovni števec se poveča.
- **VC** - vertikalno povečevanje oziroma zmanjševanje naslova. VC=0; vertikalni naslovni števec se zmanjša. VC=1; vertikalni naslovni števec se poveča.
- **HV** - smer vpisovanja podatkov. HV=0; podatki se vpisujejo kontinuirano v horizontalni smeri. HV=1; podatki se vpisujejo kontinuirano v vertikalni smeri.
- **DFM[1:0],TRI** Podajajo dolžino posameznega prenosa SPI ter število sporočil potrebnih za prenos ene slikovne točke v slikovni medpomnilnik.

DFM1	DFM0	TRI	dolžina prenosa	tip prenosa
0	0	x	18-bit	18-bitni, enojni prenos (1x 18-bit)
0	1	x	16-bit	16-bitni, enojni prenos (1x 16-bit)
1	0	x	9-bit	18-bitni, dvojni prenos (2x 9-bit)
1	1	0	8-bit	16-bitni, dvojni prenos (2x 8-bit)
1	1	1	8-bit	18-bitni, trojni prenos (3x 8-bit)

DDRAM_DATA_ACCESS_PORT (22h)

	Bits [17:12]	Bits [11:6]	Bits [5:0]
R/W	DB[17:12]	DB[11:6]	DB[5:0]
privzeto:	R	G	B

Ko naslovimo register DDRAM_DATA_ACCESS_PORT, preklopimo v način za prenos slike v slikovni medpomnilnik. Poslati moramo le ukazni del sporočila.

6.2 Nastavitev krmilnika SEPS525

Krmilnik moramo pred uporabo nastaviti s pomočjo vnaprej predpisane zagonske procedure. Zagonska procedura predpisuje vrstni red registrov, v katere je potrebno pisati, da se krmilnik postavi v začetno stanje. Nato pa lahko nastavimo vse ostale registre glede na naše potrebe. Procedura je sledeča:

- REG_SOFT_RST = 0x1 (ponastavitev na privzete vrednosti)
- REG_REDUCE_CURRENT = 0x1 (resetiranje analognega dela)
- zakasnitev 1 ms
- REG_REDUCE_CURRENT = 0x0 (normalno delovanje)
- zakasnitev 1 ms
- REG_DISP_ON_OFF = 0x0 (ugasnjen zaslon)
- REG_SOFT_RST = 0x0 (normalno delovanje)
- nastavitve vseh ostalih registrov
- REG_DISP_ON_OFF = 0x1 (prižgan zaslon)

6.3 Upravljanje krmilnika SEPS525

Da bi upravljali krmilnik SESP525, ki je povezan preko podatkovnega vodila SPI, moramo poznati način na katerega lahko to storimo znotraj jedra Linux. Jedro ponuja podatkovni strukturi "spi_transfer" in "spi_message", v nadaljevanju poimenovani kot prenos SPI in sporočilo SPI.

S podatkovno strukturo "spi_transfer" opišemo prenos, ki ga želimo opraviti na vodilu SPI. Nastaviti moramo kazalec na podatke, ki jih želimo prenesti ("tx_buf"), v primeru sprejema pa tudi kazalec na medpomnilnik ("rx_buf") za hranjenje sprejetih podatkov. Nastaviti moramo tudi dolžino besede ("bits_per_word") ter dolžino in hitrost prenosa ("count" in "spi_speed_hz").

Sporočilo SPI najprej ponastavimo s pomočjo funkcije "spi_message_init()" in mu nato dodamo enega ali več prenosov SPI. Dodajamo jih s pomočjo funkcije "spi_message_add_tail()". Sporočilo SPI nato pošljemo s pomočjo funkcije "spi_sync()". Slika 6.2 podaja primer funkcije za pošiljanje podatkov preko podatkovnega vodila SPI.

```
static int
seps525fb_spi_send(struct seps525fb *seps525fb,
                u8 *buf, size_t count, int bpw)
{
        struct spi_message  m;
        struct spi_transfer tr;

        tr.tx_buf = buf;
        tr.rx_buf = NULL;
        tr.bits_per_word = bpw;
        tr.len = count;
        tr.speed_hz = seps525fb->spi_speed_hz;

        spi_message_init(&m);
        spi_message_add_tail(&tr, &m);

        return spi_sync(seps525fb->sdev, &m);
}
```

Slika 6.2: Funkcija za pošiljanje podatkov preko podatkovnega vodila SPI, znotraj jedra Linux

Zgornjo funkcijo nato uporabimo za upravljanje krmilnika SEPS525 (dostop do registrov in prenos slik v slikovni medpomnilnik).

Krmilnik določa, da ga moramo, pred prenosom sporočila SPI, obvestiti o tem, ali prenašamo ukaze ali pa podatke. Obvestimo ga preko nožice RS, ki jo lahko upravljamo preko krmilnika GPIO. Nožice GPIO lahko nastavljamo v jedru Linux s pomočjo funkcije "gpio_set_value()". Uporabimo jo pred pošiljanjem sporočila SPI, kot to prikazuje Slika 6.3. Slednja nam prikazuje tudi funkcijo "seps525fb_write()", ki nam omogoča pisanje v registre krmilnika.

```
static inline int
seps525fb_write_cmd(struct seps525fb *seps525fb, u8 cmd)
{
        gpio_set_value(seps525fb->gpio_rs, 0);
        return seps525fb_spi_send(seps525fb, &cmd, 1, 8);
}

static inline int
seps525fb_write_dat(struct seps525fb *seps525fb, u8 dat)
{
        gpio_set_value(seps525fb->gpio_rs, 1);
        return seps525fb_spi_send(seps525fb, &dat, 1, 8);
}

static int
seps525fb_write(struct seps525fb *seps525fb, u8 cmd, u8 dat)
{
        int e;
        e  = seps525fb_write_cmd(seps525fb, cmd);
        e |= seps525fb_write_dat(seps525fb, dat);
        return e;
}
```

Slika 6.3: Funkcija za pisalni dostop do registrov krmilnika SEPS525, znotraj jedra Linux

Za prenos slike v slikovni medpomnilnik krmilnika SEPS525, moramo le nasloviti register DDRAM_DATA_ACCESS_PORT. Poslati moramo le ukazni del sporočila, kot to prikazuje Slika 6.4. Nato nastavimo nožico RS na vrednost '1' in tako povemo krmilniku, da bomo prenašali podatke oziroma sliko. Pošiljamo jo po delih, saj preko protokola SPI, ne moremo poslati poljubno dolgih sporočil. Omejili smo se na 1024 bajtov. Na koncu samo še nastavimo nožico RS nazaj na vrednost '0' (neobvezno, toda priporočljivo).

```
static int
seps525fb_write_image(struct seps525fb *dev, u8 *buf, int size)
{
        int e;

        seps525fb_write_cmd(dev, REG_DDRAM_DATA_ACCESS_PORT);
        gpio_set_value(dev->gpio_rs, 1);

        while (size) {
                count = (1024 > size) ? size : 1024;

                e = seps525fb_spi_send(dev, buf, count, dev->bpp);
                if (e) break;

                size -= count;
                buf  += count;
        }

        gpio_set_value(dev->gpio_rs, 0);
        return e;
}
```

Slika 6.4: Funkcija za prenos slike v slikovni medpomnilnik krmilnika SEPS525

6.4 Opis gonilniškega vmesnika

Jedro linux zelo natančno določa vmesnik za izdelavo gonilnikov slikovnega medpomnilnika. Določa podatkovni strukturi "fb_ops"in "fb_info", kateri moramo pravilno nastaviti in ju prijaviti jedru Linux. Okrnjeno strukturo "fb_ops" prikazuje Slika 6.5, saj za izdelavo najbolj preprostega gonilnika zadoščata že datotečni operaciji "fb_read"in "fb_write".

```
struct fb_ops {
        ssize_t (*fb_read)(struct fb_info *info,
                           char __user *buf,
                           size_t count, loff_t *ppos);
        ssize_t (*fb_write)(struct fb_info *info,
                            const char __user *buf,
                            size_t count, loff_t *ppos);
};
```

Slika 6.5: Okrnjena podatkovna struktura "fb_ops"

Slika 6.6 prikazuje poenostavljeno podatkovno strukturo "fb_info", ki med drugimi podatki vsebuje tudi statične in dinamične informacije o slikovnem medpomnilniku, kazalec na podatkovno strukturo tipa "fb_ops"("fbops"), kazalec na interni slikovni medpomnilnik ("screen_base") ter podatek o njegovi velikosti ("screen_size").

```
struct fb_info {
        struct fb_var_screeninfo var;
        struct fb_fix_screeninfo fix;
        struct fb_ops *fbops;
        char __iomem *screen_base;
        unsigned long screen_size;
};
```

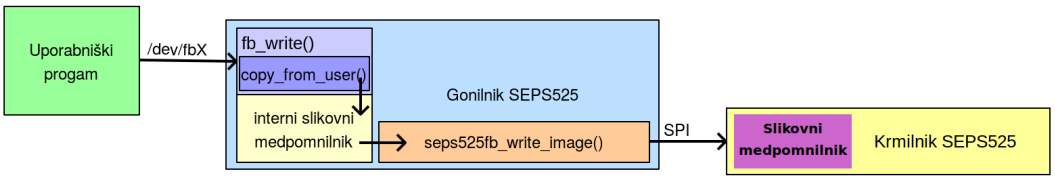
Slika 6.6: Okrnjena podatkovna struktura "fb_info"

6.5 Implementacija gonilnika SEPS525

Izdelali smo gonilnik slikovnega medpomnilnika za krmilnik SEPS525. Pri izdelavi smo uporabili že omenjene funkcije za prenos podatkov preko protokola SPI. Podprli smo samo prenos podatkov v smeri proti krmilniku, saj načrtovalci vgrajenega sistema (načrtno) niso povezali vhodne (MISO) linije podatkovnega vodila SPI.

Najprej smo napisali funkcijo "seps525fb_fb_write()", ki omogoča uporabniškim programom pisalni dostop do slikovnega medpomnilnika. Ta deluje tako, da sprejema slikovne podatke s strani uporabniškega programa, ki trenutno upravlja z napravo slikovnega medpomnilnika (/dev/fb[0..31]). Prejeti podatki so shranjeni v delu pomnilnika na katerega kaže podani kazalec "buf" in po velikosti ustrezajo podanemu parametru "count". Odmik znotraj slike pa je podan s parametrom "ppos". Okrnjeno funkcijo prikazuje Slika 6.8.

Slikovne podatke moramo najprej prenesti iz uporabniškega v jedrni način. To storimo z uporabo funkcije "copy_from_user()". S slednjo prenesemo podatke, z upoštevanjem podanega odmika, v interni slikovni medpomnilnik gonilnika. Slednjega nato, s pomočjo funkcije "seps525fb_write_image()", prenesemo v slikovni medpomnilnik krmilnika SEPS525. Slika 6.7 prikazuje prenos slikovnih podatkov od uporabniškega programa do krmilnika SEPS525.



Slika 6.7: Prikaz prenosa slikovnih podatkov od uporabniškega programa, preko gonilnika, do krmilnika SEPS525

```
static ssize_t
seps525fb_fb_write(struct fb_info *info,
                const char __user *buf,
                size_t count, loff_t *ppos)
{
        struct seps525fb *seps525fb = info->par;
        unsigned long pos = *ppos;
        u8 __iomem *dst;

        dst = (void __force *) (info->screen_base + pos);

        if (copy_from_user(dst, buf, count))
                return -EFAULT;

        seps525fb_write_image(seps525fb,
                seps525fb->fbi->fix.smem_start,
                seps525fb->fbi->fix.smem_len);

        *ppos += count;

        return count;
}
```

Slika 6.8: Okrnjena pisalna funkcija naprave slikovnega medpomnilnika

Zgornjo funkcijo bomo v nadaljevanju, preko statične podatkovne strukture "seps525fb_fbops" (Slika 6.9), prijavili jedru Linux.

```
static struct fb_ops seps525_fbops =
{
        .owner    = THIS_MODULE,
        .fb_write = seps525fb_fb_write,
};
```

Slika 6.9: Statična podatkovna struktura "seps525fb_fbops"

Jedro Linux določa, da morajo gonilniki naprav, povezanih preko podatkovnih vodil, vsebovati funkcijo "probe()". Preko nje preveri prisotnost naprav na posameznih podatkovnih vodilih in v primeru uspešno izvedene funkcije omogoči ustrezni gonilnik. Znotraj nje izvedemo tudi operacije, potrebne za nadaljne delovanje gonilnika (dinamični zaseg internih podatkovnih struktur, ponastavitev strojne opreme, prijava prekinitev jedru Linux, prijava podprtih vmesnikov jedru Linux ...).

Znotraj funkcije "probe()", prikazane na Sliki 6.10, najprej dinamično zasežemo podatkovno strukturo "fb_info", ki opisuje napravo slikovnega medpomnilnika. Slednja vsebuje podatkovni strukturi za hranjenje statičnih in dinamičnih informacij o slikovnem medpomnilniku, ki smo jih v nadaljevanju nastavili glede na lastnosti naše slikovne prikazovalne naprave (160x128; RGB565). Še pred tem zasežemo tudi interni slikovni medpomnilnik, katerega uporabljamo za vmesno hranjenje slike. Kazalec nanj shranimo v podatkovno strukturo "fb_info". Nato pokličemo funkcijo "seps525_init", s katero nastavimo krmilnik SEPS525 glede na naše potrebe. Funkcija izvede tudi zagonsko proceduro krmilnika. Nato nastavimo kazalec "info->fbops" tako, da kaže na našo statično podatkovno strukturo "seps525fb_fbops", ki podaja podprte datotečne operacije nad napravo slikovnega medpomnilnika. Na koncu le še prijavimo napravo slikovnega medpomnilnika jedru Linux, s pomočjo funkcije "register_framebuffer()".

Gonilnik smo nato prijavili jedru Linux s pomočjo statične podatkovne strukture "seps525fb_driver" in uporabo konstrukta (module_spi_driver) za registracijo gonilnikov naprav SPI. Slednje je prikazano na Sliki 6.11.

Bralca opozarjamo, da so vse prikazane podatkovne strukture in funkcije zaradi preglednosti okrnjene in so zato namenjene zgolj predstavitvi glavnih konceptov gonilnika.

```
static int
seps525fb_probe(struct spi_device *sdev)
{
        struct fb_info *info;

        info = framebuffer_alloc(size, &sdev->dev);

        vmem_size  = 160 * 128 * 2;
        vmem       = devm_kzalloc(&sdev->dev,
                        vmem_size, GFP_KERNEL);

        info->fix.id          = "SEPS525";
        info->fix.type        = FB_TYPE_PACKED_PIXELS;
        info->fix.visual      = FB_VISUAL_TRUECOLOR;
        info->fix.line_length = 160 * 2;
        info->screen_base     = vmem;
        info->fix.smem_start  = vmem;
        info->fix.smem_len    = vmem_size;

        info->var.bits_per_pixel = 16;
        info->var.xres           = 160;
        info->var.yres           = 128;
        info->var.red.length     = 5;
        info->var.red.offset     = 0;
        info->var.green.length   = 6;
        info->var.green.offset   = 5;
        info->var.blue.length    = 5;
        info->var.blue.offset    = 11;

        seps525_init();

        info->fbops = &seps525_fbops;
        register_framebuffer(info);
}
```

Slika 6.10: Okrnjena funkcija ”probe()” gonilnika SEPS525

```
static struct spi_driver seps525fb_driver =
{
        .probe  = seps525fb_probe,
        .remove = seps525fb_remove,
        .driver =
        {
                .name  = "seps525fb",
                .owner = THIS_MODULE,
        },
};

module_spi_driver(seps525fb_driver);
```

Slika 6.11: Prijava gonilnika jedru Linux

Poglavje 7

Razvoj pomožnih orodij

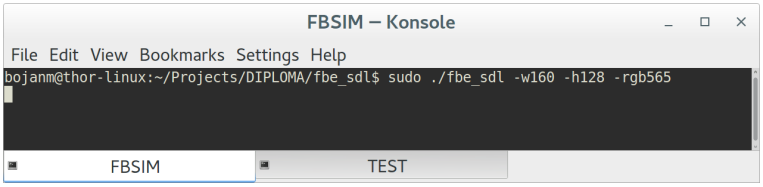
Razvili smo sledeča pomožna orodja s katerimi smo pohitrili razvoj uporabniškega vmesnika:

- simulator slikovnega medpomnilnika
- orodje za pretvorbo obrisnih pisav v bitne

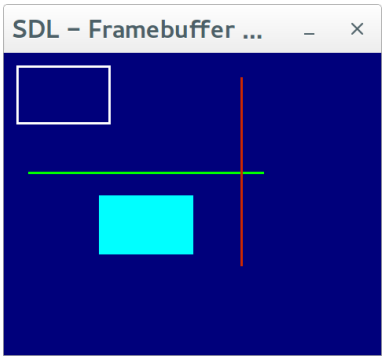
7.1 Simulator slikovnega medpomnilnika

Testiranje programskih knjižnic bi bilo zelo počasno, v kolikor bi morali za vsako spremembo knjižnice prenesti testni program na vgrajeno napravo. Posledično smo se odločili izdelati simulator slikovnega medpomnilnika, preko katerega smo lahko veliko hitreje odkrivali napake v programski kodi. Izdelali smo ga s pomočjo programke knjižnice SDL2 [25]. Najprej smo napisali programsko kodo, ki odpre izrisovalno okno SDL (Slika 7.2) glede na podane argumente znotraj ukazne lupine Linux. Možno je nastaviti velikost prikazovalnega okna ter izbrati barvni model RGB (Slika 7.1).

Simulator je zasnovan na podlagi deljenega pomnilnika (ang. shared memory), ki se ob zagonu programa ustvari v obliki deljene datoteke /var/run/fb_screen[0..31]. Slednja po velikosti ustreza količini podatkov, potrebnih za prikaz ene slike. Simulator nato le še neprestano izrisuje vsebino te datoteke znotraj prikazovalnega okna. Testni program pa namesto naprave



Slika 7.1: Poganjanje simulatorja slikovnega medpomnilnika



Slika 7.2: SDL izrisovalno okno simulatorja slikovnega medpomnilnika

slikovnega medpomnilnika (/dev/fb[0..31]) odpre našo deljeno datoteko in jo s pomočjo sistemskega klica mmap() preslika v svoj pomnilniški naslovni prostor. Tako lahko le z uporabo pomnilniškega pisanja izrisuje slike znotraj prikazovalnega okna simulatorja.

7.2 Orodje za pretvorbo obrisnih pisav

Naredili smo orodje za pretvorbo obrisnih pisav TrueType [26, 7] v bitne pisave. Pri izdelavi smo uporabili obstoječo programsko knjižnico FreeType [20], katere namen je upodabljanje obrisnih pisav. Knjižnici podamo izbrano pisavo in ji povemo, na kakšen način jo želimo upodobiti. V orodju smo podprli tri različne upodobitve:

- **FT_RENDER_MODE_MONO** - upodobitev pisav brez glajenja, kjer so glifi oziroma posamezni znaki pisave upodobljeni kot bitne slike.

- **FT_RENDER_MODE_NORMAL** - upodobitev pisav z uporabo metode glajenja robov, kjer so glifi oz. posamezni znaki pisave upodobljeni kot sivinske slike intenzitet.
- **FT_RENDER_MODE_LCD** - upodobitev pisav z uporabo podtočkovnega glajenja, glajenja robov, kjer so glifi oziroma posamezni znaki pisave upodobljeni kot sivinske slike intenzitet treh barvnih komponent (rdeče, zelene in modre). Slika 7.3 prikazuje poenostavljen primer upodobitve obrisne pisave v tem načinu.

```
FT_Init_FreeType(&library);

FT_New_Face(library, "FreeSansMono.ttf", 0, &face);
FT_Set_Char_Size(face, size << 6, 0, dpi, 0);

for (c = first; c <= last; c++) {
        FT_Load_Char(face, c, FT_LOAD_TARGET_LCD);
        FT_Render_Glyph(face->glyph, FT_RENDER_MODE_LCD);
        FT_Get_Glyph(face->glyph, &glyph);
        glyph_store(&glyph);
        FT_Done_Glyph(glyph);
}
save_glyphs();

FT_Done_FreeType(library);
```

Slika 7.3: Poenostavljen primer pretvorbe obrisne pisave s podtočkovni glajenjem

Najprej izberemo obrisno pisavo in določimo velikost, v kateri jo želimo upodobiti. Pomik v levo je potreben zaradi tega, ker knjižnica uporablja predstavitev števil v fiksni vejici, formatu 26.6 (6 decimalnih mest). Nato moramo znotraj zanke upodobiti vsak znak pisave posebej ter ga shraniti v lastne, predhodno alocirane podatkovne strukture. V našem primeru to

nalogo opravi funkcija glyph_store(). Na koncu moramo le še na ustrezen način zapisati upodobljene slike znakov v datoteko. Pri nas to opravi funkcija save_glyphs(). Odločili smo se, da jih zapišemo v obliki statične podatkovne strukture, kot izvorno datoteko programskega jezika C. Tako pretvorjene pisave smo nato vključili kot del programske kode v grafično programsko knjižnico, ki jih uporablja za izris teksta.

Poglavje 8

Razvoj grafičnega uporabniškega vmesnika

V skladu z že narejenim načrtom v poglavju 5 smo izdelali grafični uporabniški vmesnik za Xiphra SE šifrirne naprave. Celoten problem izdelave smo razbili v več manjših, neodvisnih ter bolj obvladljivih nalog. V tem poglavju se bomo na kratko dotaknili vsake izmed teh nalog ter na koncu pokazali, kako smo jih združili v končno rešitev.

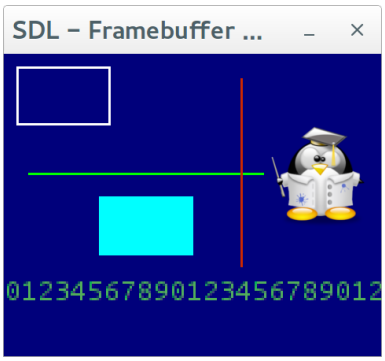
Izdelana je bila sledeča programska oprema:

- grafična programska knjižnica
- programska knjižnica gradnikov uporabniškega vmesnika
- grafični uporabniški vmesnik za naprave Xiphra SE

8.1 Grafična programska knjižnica

Izdelali smo grafično programsko knjižnico, ki smo jo kasneje uporabili za izris gradnikov uporabniškega vmesnika. Napisali smo algoritme za izris osnovnih geometrijskih oblik, kot so točka, horizontalna črta, vertikalna črta, pravokotnik ter zapolnjen pravokotnik, ki so predstavljeni v nadaljevanju. Dodali smo še funkciji za izris teksta in slike. Vse izrisovalne funkcije pa smo na

koncu še dodatno nadgradili s podporo za grafično obrezovanje (ang. clipping). Slika 8.1 prikazuje uporabo vseh zgoraj naštetih funkcij.



Slika 8.1: Primer uporabe grafične programske knjižnice

Funkcije izrisujejo v interni slikovni medpomnilnik, slika pa se zapiše v napravo slikovnega medpomnilnika šele ob klicu funkcije "flip()". Tako preprečimo nezaželeno utripanje slike ter tudi pohitrimo izrisovanje, ki je dodatno pohitreno tudi tako, da imamo za vsako barvno globino svoj nabor zgoraj naštetih funkcij (hline8, hline16, rect8, rect16, ...). S tem pristopom je možno izkoristiti prednosti širših podatkovnih vodil CPE (16 oz. 32-bitno pomnilniško pisanje namesto 8-bitno).

8.1.1 Izris točke

Točka je v dvodimenzionalnem prostoru podana s koordinatama x in y. Če jo želimo izrisati moramo najprej izračunati njen odmik znotraj slikovnega medpomnilnika, kot to prikazuje enačba 8.1. Širina v enačbi predstavlja horizontalno zaslonko ločljivost. Barvna globina pa je podana v bitih.

$$odmik = (y * sirina + x) * (barvna\_globina/8) \tag{8.1}$$

Nato z izračunanim odmikom dostopamo do slikovnega medpomnilnika in vanj zapišemo barvno vrednost, kot to prikazuje Slika 8.2.

```
void putpixel(struct fbgfx *gc, s32 x, s32 y, u32 color)
{
        u32 offset = (y * gc->info.width + x) *
                      gc->info.bytes_per_pixel;

        gc->bbuffer[offset] = color & gc->color_mask;
}
```

Slika 8.2: Grafična funkcija za izris točke

8.1.2 Izris horizontalne črte

Horizontalno črto izrišemo tako, da najprej izračunamo odmik prve slikovne točke znotraj slikovnega medpomnilnika, nato pa zapišemo barvno vrednost v ustrezno število sosednjih slikovnih točk (memset16), kot to prikazuje Slika 8.3.

```
void hline16(struct fbgfx *gc, s32 x, s32 y,
                u16 length, u32 color)
{
        u32 offset = (y * gc->info.width + x) * 2;
        u8 *address = gc->bbuffer + offset;
        memset16(address, color & 0xFFFF, length);
}
```

Slika 8.3: Grafična funkcija za izris horizontalne črte (16 bitna barvna globina)

8.1.3 Izris vertikalne črte

Vertikalno črto izrišemo tako, da najprej izračunamo odmik prve slikovne točke znotraj slikovnega medpomnilnika. Nato pa se pomikamo po y osi s prištevanjem dolžine zaslonske vrstice (line_size). Pred vsakim premikom vpišemo barvno vrednost. Algoritem prikazuje Slika 8.4.

```
void vline16(struct fbgfx *gc, s32 x, s32 y,
                u16 length, u32 color)
{
        u32 offset = (y * gc->info.width + x) * 2;

        for (int i=0; i < length; i++) {
                gc->bbuffer[offset] = color & 0xFFFF;
                offset += gc->info.line_length;
        }
}
```

Slika 8.4: Grafična funkcija za izris vertikalne črte (16 bitna barvna globina).

8.1.4 Izris zapolnjenega pravokotnika

Zapolnjen pravokotnik izrišemo s pomočjo funkcije za izris horizontalne črte (Slika 8.5). Vhodni parameter rect podaja pravokotnik, ki ga moramo narisati. Podaja koordinati x in y ter dolžino (rect->w) in širino (rect->h).

```
void fillrect16(struct fbgfx *gc, fbgfx_rect_t *rect, u32 color)
{
        for (s32 yi=rect->y; yi < (rect->y + rect->h); yi++)
                hline16(gc, rect->x, yi, rect->w, color);
}
```

Slika 8.5: Grafična funkcija za izris zapolnjenega pravokotnika (16 bitna barvna globina)

8.1.5 Izris pravokotnika

Pravokotnik izrišemo z uporabo funkcij za izris horizontalne in vertikalne črte, kot to prikazuje Slika 8.6. Vhodni parameter rect podaja pravokotnik, ki ga moramo narisati. Podaja koordinati x in y ter dolžino (rect->w) in širino (rect->h).

```
void rect16(struct fbgfx *gc, fbgfx_rect_t *rect, u32 color)
{
        s32 x1 = rect->x;
        s32 y1 = rect->y;
        s32 x2 = rect->x + rect->w - 1;
        s32 y2 = rect->y + rect->h - 1;

        hline16(gc, x1, y1, rect->w, color);
        hline16(gc, x1, y2, rect->w, color);
        vline16(gc, x1, y1, rect->h, color);
        vline16(gc, x2, y1, rect->h, color);
}
```

Slika 8.6: Grafična funkcija za izris pravokotnika (16 bitna barvna globina).

8.1.6 Izris teksta

Bitna pisava je predstavljena s podatkovno strukturo "fbgfx_font", kot jo prikazuje Slika 8.8. Struktura hrani kazalec na bitno sliko (bitmap) v kateri so zaporedoma shranjene bitne slike posameznih znakov. Elementa first in last nam podata obseg znakov, ki so nam na voljo (ASCII koda prvega in zadnjega vsebovanega znaka) v seznamu glyph. Slednji vsebuje vse potrebne informacije za predstavitev posameznega znaka: odmik znotraj bitne slike (bitmap_offset), širino in višino (width in height), potreben premik po x osi po izrisanem znaku (xadvance) ter odmik prve slikovne točke znaka (xoffset in yoffset). Podatkovna struktura, ki vsebuje omenjene informacije, je prikazana na Sliki 8.7.

```
typedef struct fbgfx_glyph
{
        u16 bitmap_offset;
        u8  width;
        u8  height;
        u8  xadvance;
        s8  xoffset;
        s8  yoffset;
}
fbgfx_glyph_t;
```

Slika 8.7: Podatkovna struktura za predstavitev glifa oziroma znaka znotraj bitne pisave

```
typedef struct fbgfx_font
{
        int type;
        u8 *bitmap;
        fbgfx_glyph_t *glyph;
        u8 first;
        u8 last;
        u8 yadvance;
}
fbgfx_font_t;
```

Slika 8.8: Podatkovna struktura za predstavitev bitne pisave

Vsak znak bitne pisave je predstavljen kot bitna oziroma v primeru glajenja kot sivinska slika intenzitet. Tekst izrišemo tako, da se sprehodimo po podanem nizu in izrisujemo posamezne glife oziroma znake. Izrisujemo vsako točko posamezno, saj moramo v primeru glajene pisave (type $>$ 0) ustrezno prilagajati intenzitete posameznih točk. Po izrisu vsakega znaka se premaknemo v desno za xadvance slikovnih točk.

8.1.7 Izris slike

Slike so predstavljene s preprosto strukturo, ki jo prikazuje Slika 8.10. Struktura podaja njeno širino, dolžino, barvno globino, izraženo v bajtih, ter kazalec na slikovne podatke. Sliko izrišemo tako, da jo prenesemo vrstico po vrstico v slikovni medpomnilnik, kot to prikazuje Slika 8.10. Barvni model slike mora ustrezati barvnemu modelu slikovnega medpomnilnika.

```
typedef struct fbgfx_image
{
        u16 width;
        u16 height;
        u32 bytes_per_pixel;
        u8 *data;
}
fbgfx_image_t;
```

Slika 8.9: Podatkovna struktura za predstavitev slik

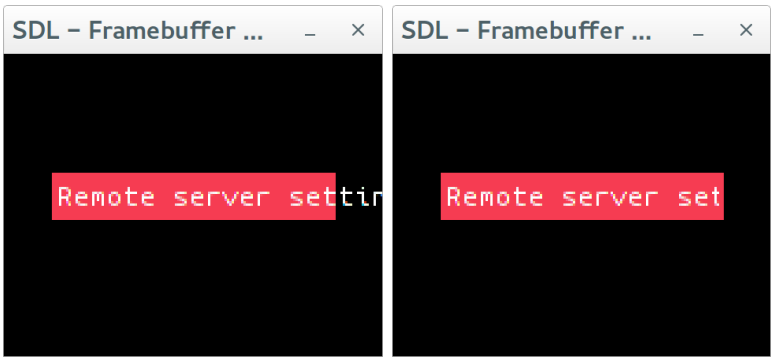
```
int fbgfx_draw_image16(struct fbgfx *gc, s32 x, s32 y,
                       struct fbgfx_image *img)
{
        offset  = (y * gc->info.width + x) * 2;
        bbuffer = gc->bbuffer + offset;
        data    = img->data;

        for (i=0; i < img->height; i++) {
                memcpy16(bbuffer, data, img->width);
                bbuffer += gc->info.line_length;
                data    += img->width * 2;
        }
}
```

Slika 8.10: Poenostavljena funkcija za izris slike (16 bitna barvna globina)

8.1.8 Grafično obrezovanje:

Grafično obrezovanje (ang. clipping) je metoda, s katero lahko omejimo področje izrisovanja. Je ključnega pomena za izris grafičnih uporabniških vmesnikov, saj nam omogoča, da omejimo področje izrisa na področje znotraj posameznih gradnikov. Za lažje razumevanje si poglejmo Sliko 8.11, na kateri smo izrisali dva gumba, enega z grafičnim obrezovanjem (desno) in enega brez (levo). Na slednjem se je njegov naslov oziroma tekst izrisal izven njegovih meja.



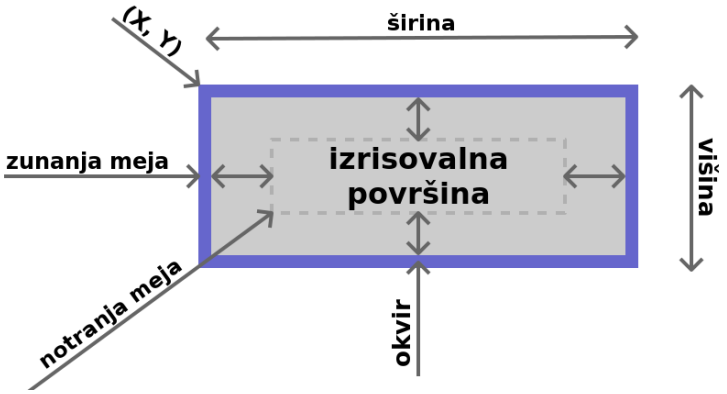
Slika 8.11: Primer grafičnega obrezovanja pri izrisu gumba. Gumb v desnem oknu je imel nastavljeno obrezovalno okno, v levem pa ne

Grafično obrezovanje smo zasnovali tako, da smo v vsako izrisovalno funkcijo dodali dodatno preverjanje robnih koordinat ter jih prilagodili tako, da niso presegale nastavljenega obrezovalnega okna.

8.2 Programska knjižnica gradnikov uporabniškega vmesnika

Izdelana je bila programska knjižnica za izris gradnikov uporabniškega vmesnika. Knjižnica je napisana v programskem jeziku C in uporablja določene pristope, ki so bolj značilni za objektni način programiranja. Vsi gradniki (ang. widgets) so namreč predstavljeni s svojo podatkovno strukturo, ki vsebuje gradniku lastne podatke ter kazalce na funkcije oziroma metode. S takšnim pristopom smo se približali objektnemu programiranju, s katerim smo tudi poenostavili vmesnik knjižnice (API). Vsak gradnik je znotraj knjižnice predstavljen kot podrazred glavnega razreda widget. Slednji vsebuje skupne lastnosti vseh gradnikov in je najpomembnejši, saj določa osnovni programski vmesnik za vse ostale razrede (Dodatek A). Glavni razred določa, da je vsak gradnik omejen z zunanjimi mejami v obliki pravokotnika (ang. bounds). Pravokotnik znotraj zunanje meje pa je lahko zapolnjen in obrobljen s poljubno debelim okvirjem. Določa tudi odmike znotraj zunanje meje (ang. margins), ki pravzaprav določajo notranjo mejo gradnika. Slednja predpisuje izpeljanim gradnikom njihovo izrisovalno površino. Vsakemu gradniku je možno spremeniti položaj, ga povečati oziroma zmanjšati, spremeniti barve ter mu določiti pisavo za izpis teksta. Za lažjo predstavo si poglejmo Sliko 8.12.

Vsak izpeljani gradnik mora vsebovati funkcijo process_input_event() preko katere bo sprejemal vhodne dogodke ter se ustrezno na njih odzival. Ob sprejemu vhodnega dogodka ga lahko zavrže ali pa se odzove nanj tako, da spremeni svoj izris (spremeni barvo, doda nov znak, ...). Vsebovati mora tudi funkcijo draw(), preko katere se bo na našo zahtevo izrisal. Glavni razred določa, da ima vsak gradnik tri nastavljiva stanja: omogočen, neomogočen ter fokusiran. Gradnik lahko sprejema vhodne dogodke, le kadar je omogočen in fokusiran. Ko je fokusiran, se za njegov izris uporabi drug nabor barv oz stil (nastavljivo). Ko pa je onemogočen, pa mora biti izrisan v sivinskih odtenkih.



Slika 8.12: Osnovne lastnoti gradnikov

Izpeljani razredi razreda widget:

- gumb (razred wbutton)
- tekstovno polje (razred wlabel)
- vnosno tekstovno polje (razred winput)
- navidezna tipkovnica (razred wkeyboard)
- meni (razred wmenu)
- ikonski meni (razred wiconmenu)
- stran (razred wpage)

Izpeljanim razredom lahko določimo povratne klice oziroma funkcije (ang. callbacks), ki se bodo poklicale ob ustreznih vhodnih dogodkih. Gumbu lahko na primer določimo povratni klic, ki se bo izvedel ob pritisku nanj. Vnosnemu polju lahko določimo povratni klic ob zaključenem vnosu. Vsakemu gradniku pa lahko določimo povratni klic, ko pridobi ali izgubi fokus. Takšnih klicev je še mnogo več. Z njihovo pomočjo bomo lahko kasneje upravljali uporabniški vmesnik.

Z izjemo razreda wpage smo vse gradnike opisali že v fazi načrtovanja (poglavje 5). Slednjega nismo načrtovali, ga pa potrebujemo. Gradnik wpage

namreč ponuja navigacijo med vsebovanimi gradniki (wlabel, winput, wbutton ...), ki mu jih lahko poljubno dodajamo ali odvzemamo. Gradniki, ki jih vsebuje so lahko izven meja izrisovalne površine (niso izrisani). Ob sprejemu ustreznega vhodnega dogodka bo izbral naslednji oziroma prejšnji vsebovan gradnik ter prilagodil položaje vseh vsebovanih gradnikov tako, da bo izbrani gradnik viden.

Pri načrtovanju programske knjižnice smo pazili na zastavljene cilje, ki smo jih na koncu tudi uspešno dosegli:

- majhna poraba delovnega pomnilnika
- statično zaseganje pomnilnika
- delovanje brez prisotnosti operacijskega sistema
- podpora za krožni kodirnik
- podpora za navidezno tipkovnico
- enostavni gradniki

Uporabnikom programske knjižnice smo omogočili izklop prevajanja posameznih gradnikov, v kolikor jih v svoji končni rešitvi ne bodo potrebovali. Posledično lahko povečujemo oziroma zmanjšujemo potrebo po delovnem pomnilniku, ki pa je že v osnovi majhna. Omogočili smo tako dinamično kot tudi statično zaseganje pomnilnika (izbira možna ob prevajanju). Neodvisnost od operacijskega sistema smo dosegli z uporabo splošnonamenskega vmesnika za delo z vhodnimi dogodki. Podprli smo krožni kodirnik in omogočili izpolnjevanje vnosnih polj preko navidezne tipkovnice. Pazili pa smo tudi na enostavnost gradnikov ter jih zasnovali tako, da so lahko izrisani na majhnih slikovnih prikazovalnikih. Izgled gradnikov bo razviden v naslednjem podpoglavju, kjer smo jih uporabili pri izdelavi uporabniškega vmesnika. Še prej pa si poglejmo pregled lastnosti naše knjižnice (Tabela 8.1) in jo primerjajmo z obstoječimi rešitvami (Tabela 8.2). Vidimo lahko, da smo zares izpolnili vse zastavljene cilje, saj smo odpravili vse pomankljivosti obstoječih rešitev.

libwidget	
programski jezik:	C
operacijski sistem:	Linux
samostojno delovanje:	DA, OS ni potreben
poraba delovnega pomnilnika:	$<$ 4MB
statično zaseganje pomnilnika:	DA
slikovni medpomnilnik:	DA
podprte zaslonske ločljivosti:	ni posebnih omejitev
barvni formati:	RGB565, RGB666, RGB888, sivinski, monokromatski, ...
vsi gradniki up. vmesnika:	DA
navidezna tipkovnica:	DA
enostavni gradniki:	DA
profesionalen izgled gradnikov:	**ocena bi bila subjektivna** [1]
navadno glajenje teksta:	DA
podtočkovno glajenje teksta:	DA
podpora za krožni kodirnik:	DA
dostopna izvorna koda:	DA
kvalitetna izvorna koda:	**ocena bi bila subjektivna** [2]
strošek izdelave:	100 ur

Tabela 8.1: Pregled lastnosti lastne knjižnice gradnikov uporabniškega vmesnika

[1] Menimo, da izgled gradnikov povsem zadošča namembnosti uporabniškega vmesnika ter smo z izgledom zadovoljni.

[2] Menimo, da smo napisali kvalitetno in učnkovito programsko kodo, saj imamo na tem področju dolgoletne izkušnje.

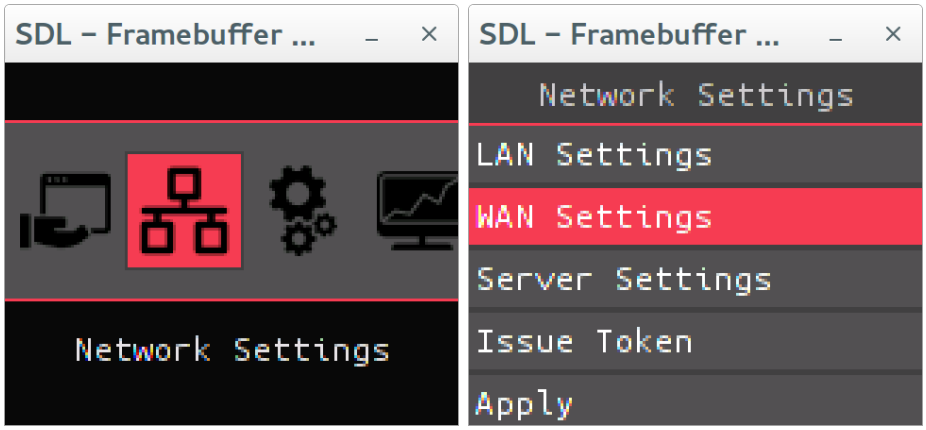
	Qt Embedded	**easyGUI**	**μGFX**	**libwidget**
programski jezik C		x	x	x
sistem Linux	x	x	x	x
samostojno delovanje		x	x	x
poraba pomnilnika		x	x	x
statično zaseganje		/		x
slikovni medpomnilnik	x	x	x	x
zaslonske ločljivosti	x	x	x	x
barvni formati	x	x	x	x
vsi gradniki	x	x	x	x
navidezna tipkovnica				x
enostavni gradniki		x	x	x
izgled gradnikov	x		x	x
glajenje teksta	x	x	x	x
podtočkovno glajenje	x			x
podprt krožni kodirnik			x	x
dostopna izvorna koda	x		x	x
kvalitetna izvorna koda	x	/		x
strošek nakupa	x		x	x

Tabela 8.2: Primerjava obstoječih rešitev napram lastni programski knjižnici gradnikov uporabniškega vmesnika

8.3 Grafični uporabniški vmesnik za naprave Xiphra SE

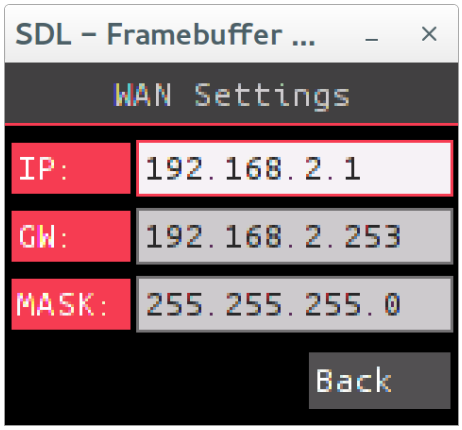
Grafični uporabniški vmesnik smo naredili s pomočjo lastne programske knjižnice gradnikov uporabniškega vmesnika. Jedro uporabniškega vmesnika je neskončna zanka, ki prikazuje trenutno izbran gradnik ter mu podaja vhodne dogodke (dogodkovni interaktivni vmesnik). Gradnike smo predhodno ustvarili, določili vse njihove lastnosti ter jim nastavili ustrezne povratne klice (kazalce na funkcije). Znotraj slednjih smo bodisi spreminjali lastnosti gradnikov bodisi izbrali nov gradnik za prikaz. To nam je omogočilo sprehajanje med gradniki oziroma navigacijo po uporabniškem vmesniku.

Skladno z načrtom smo naredili glavni ikonski meni in podmenije za izbiro podsklopov. Slika 8.13 prikazuje glavni meni in enega izmed podmenijev. Znotraj obeh se premikamo z vrtenjem gumba krožnega kodirnika, ob pritisku nanj pa se sproži ustrezni povratni klic, ki poda enoznačno oznako, s katero lahko ugotovimo uporabnikovo izbiro. Na primer oznaka prve ikone v glavnem meniju je vrednost 0, druge 1 in tako dalje. Enako je v primeru podmenija. Na podlagi te oznake se odločimo, kateri gradnik moramo izbrati za naslednji prikaz.



Slika 8.13: Prikaz glavnega menija (levo) in podmenija za izbiro sklopa mrežnih nastavitev (desno)

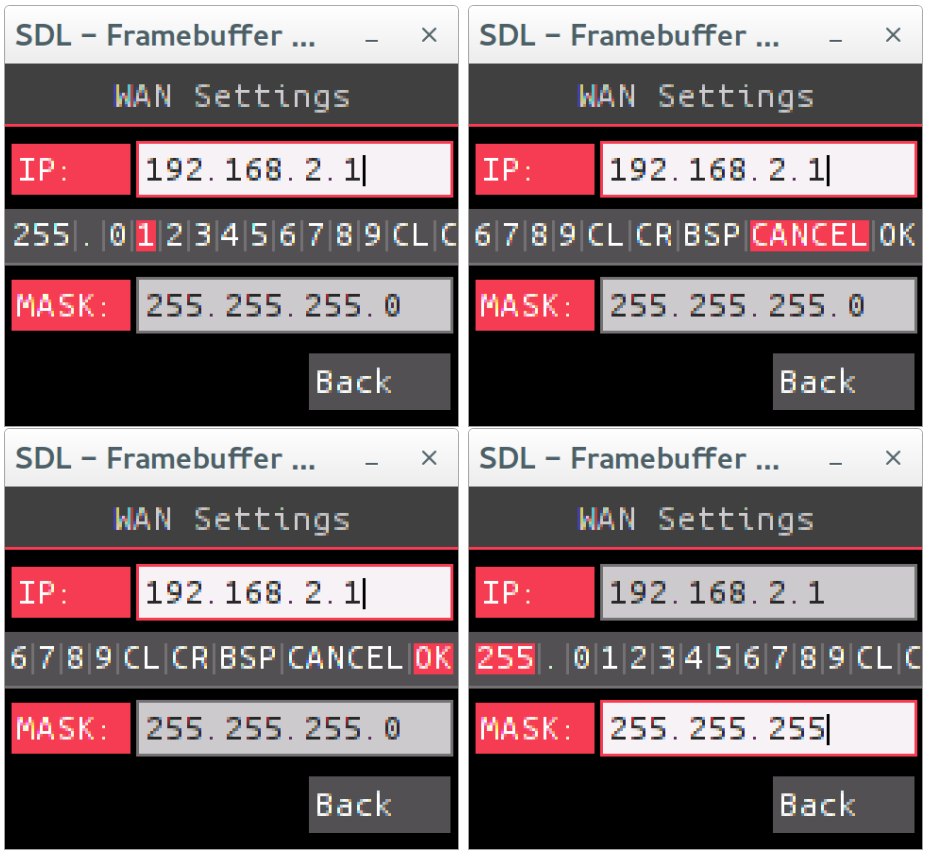
Sklopi za nastavljanje mrežnih nastavitev so zasnovani s pomočjo gradnika wpage. Vsak sklop je poimenovan, saj mora uporabnik v vsakem trenutku poznati svoj položaj znotraj vmesnika. Ti sklopi vsebujejo le tri vnosna polja in gumb za povratek nazaj (Slika 8.14). Vnašamo lahko naslov IP, naslov prehoda ter mrežno masko. Med vnosnimi polji se sprehajamo z vrtenjem gumba, s pritiskom nanj pa izberemo enega izmed njih. Za razliko od navadnih uporabniških vmesnikov, kjer je vnos možen že v trenutku, ko vnosno polje pridobi fokus, moramo v našem primeru vnosno polje še dodatno izbrati. To je ena izmed rešitev za lažjo uporabo omejene vhodne naprave.



Slika 8.14: Prikaz enega izmed sklopov za nastavljanje mrežnih nastavitev

Ob izbiri vhodnega polja se na zaslonu pojavi navidezna tipkovnica, kot jo prikazuje Slika 8.15. Je še v prototipni fazi, z njo pa vrednotimo njeno funkcionalnost. Ko bo dokončno razvita bodo nekatere navidezne tipke zamenjane z ikonami, ki bodo odstranile vse dvoumnosti (pomen tipk CL, CR in BSP). Po njenih navideznih tipkah se lahko premikamo z vrtenjem krožnega kodirnika v levo oziroma desno stran, izberemo pa jo s pritiskom na gumb. Ob izbiri se vnosnemu polju pošlje en ali več vhodnih dogodkov. Na Sliki 8.15 lahko v spodnjem desnem prikazovalnem oknu opazimo, da je možno vnesti število 255 v enem koraku (zelo pogost vnos pri vnašanju mrežnih mask). Na ta način smo uporabniku našega vmesnika pohitrili eno

izmed ponavljujočih se operacij. Po vnosnem polju se lahko premikamo z navideznima tipkama 'CL' (levo, ang. Cursor Left) ter 'CR' (desno, ang. Cursor Right). Brisanje je omogočeno prek navidezne tipke BSP (ang. backspace). Vnašanje pa zaključimo z izbiro navidezne tipke 'OK' ali preko tipke za prekinitev vnosa 'CANCEL'. Uporabnikom smo s pomočjo navidezne tipkovnice in naprednih vhodnih polj preprečili napačen vnos podatkov. Navidezna tipkovnica prikaže le znake, ki so veljavni za določen tip vnosnega polja (tekst, številke, IP naslovi, ...). Vnosno polje pa pred uporabo vhodnega dogodka preveri, če bi njegova uporaba vodila v napačen vnos in ga v tem primeru tudi zavrže (npr. napačen IP naslov, predolg vnos, preveliko število).



Slika 8.15: Prikaz uporabe navidezne tipkovnice

Slika 8.16 prikazuje uporabniški vmesnik na Xiphra SE šifrirni napravi.

Slika 8.16: Prikaz uporabniškega vmesnika na Xiphra SE šifrirni napravi

Poglavje 9

Sklepne ugotovitve

V tej diplomski nalogi smo predstavili načrtovanje in izdelavo uporabniškega vmesnika za vgrajene sisteme. Začeli smo s spoznavanjem vhodno-izhodnih naprav in se nato spoznali tudi z načinom njihove uporabe znotraj operacijskega sistema Linux. Naučili smo se osnov uporabniško usmerjenega načrtovanja uporabniških vmesnikov ter slednje razširili z upoštevanjem specifičnosti vgrajenih sistemov. S predhodno narejenim načrtom je bil tako uspešno izdelan prvi računalniški prototip uporabniškega vmesnika za Xiphra SE šifrirne naprave. Z vrednotenjem uporabniškega vmesnika s strani strank bomo dobili odziv na morebitne pomankljivosti, ki jih bomo odpravili v naslednjih iteracijah in tako še izboljšali uporabniško izkušnjo.

Izjemno smo zadovoljni z doseženim, saj smo v omejenem času izdelali številne programske komponente, ki so hkrati tudi kvalitetne ter enostavno razširljive. Zadovoljni smo tudi z dejstvom, da smo bralcu te diplomske naloge, predstavili izdelavo uporabniškega vmesnika za vgrajene sisteme na celovit in poučen način. Ponosni smo tudi na to, da smo razvili programska orodja, ki so nam znatno skrajšala potreben čas za razvoj uporabniškega vmesnika (stroškovna učinkovitost). Predvsem s pomočjo simulatorja slikovnega medpomnilnika, ki nam je omogočil hitrejše testiranje programskih knjižnic ter uporabniškega vmesnika. Uporabili pa smo ga tudi za testiranje barvnih formatov, ki jih krmilnik SEPS525 ne podpira (na primer RGB444).

Izdelali smo gonilnik slikovnega medpomnilnika za krmilnik SEPS525. Slednji je v popolnosti razvit in zato ne pričakujemo potrebe po njegovi razširitvi. Testiranje pa ni odkrilo napak ali nezanesljivosti v delovanju.

Razvita grafična programska knjižnica dostopa do gonilnika slikovnega medpomnilnika preko naprave slikovnega medpomnilnika ("fbdev"). Knjižnica podpira številne barvne formate, katerih nabor je možno enostavno razširiti. Slednje je zelo pomembno, saj se bomo v prihodnosti zagotovo srečali s slikovnimi prikazovalniki, katerih barvni formati trenutno niso podprti. Knjižnico imamo namen razširiti tudi z dodatnimi izrisovalnimi algoritmi (krog, črta, pravokotnik z zaobljenimi robovi ...).

Grafično programsko knjižnico smo uporabili znotraj programske knjižnice za izris gradnikov uporabniškega vmesnika. Slednja nam ponuja izris številnih gradnikov, ki pa jih imamo namen še dodatno razširiti. Izdelali bi lahko na primer vsaj še sledeče: seznam (ang list), potrditveno polje (ang. checkbox), večvrstično tekstovno polje (ang. text area), dialog, histogram in animacijo.

Prednost uporabe lastnih podpornih knjižnic je prilagodljivost lastnim potrebam ter specifičnosti naših naprav. Izdelane knjižnice zadoščajo vsem našim zahtevam in so povsem zadoščale za izdelavo uporabniškega vmesnika za Xiphra SE šifrirne naprave. Prednost se je pokazala tudi v možnosti tesne integracije navidezne tipkovnice z ostalimi gradniki, saj smo na tak način omogočili enostavno upravljanje uporabniškega vmesnika preko omejene vhodne naprave (krožnega kodirnika). Slabost pa je odsotnost integriranega razvojnega ogrodja za izdelavo uporabniških vmesnikov, kar se odraža v počasnejšemu razvoju. Slednje imamo namen omiliti s postavitvijo gradnikov znotraj XML datoteke, ki jo bo uporabniški vmesnik uporabil za izris gradnikov. S tem pristopom bomo tudi ločili podatkovni in prikazovalni model vmesnika.

Uporabniški vmesnik imamo namen razširiti še z vtičnim (ang. socket) protokololom, preko katerega bodo lahko ostali uporabniški procesi prikazovali sporočila znotraj uporabniškega vmesnika.

Literatura

[1] Wilbert O. Galitz. The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques, Third Edition. Wiley Publishing, 2007.

[2] Murray, James D., Van Ryper, William. Encyclopedia of Graphics File Formats, 2nd Edition. O'Reilly Media, 1996.

[3] Charles Platt. Encyclopedia of Electronic Components Volume 1. O'Reilly 2013.

[4] P. Raghavan, Amol Lad, Sriram Neelakandan. Embedded Linux System Design and Development. Auerbach Publications, 2015.

[5] Dan Saffer. Designing Gestural Interfaces: Touchscreens and Interactive Devices. O'Reilly 2009.

[6] Larry Constantine Acton, Mass. Principles of User-Interlace Design, User-Interface Design for Embedded Systems [Online]. Dosegljivo: http://m.eet.com/media/1171765/user-interface%20design%20for%20embedded%20systems.pdf [Dostopano: 26.5.2016].

[7] Janez Goričan. Pisave TrueType. Diplomsko delo, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, 2012 [Online]. Dosegljivo: https://dk.um.si/Dokument.php?id=41298 [Dostopano: 14.5.2016].

[8] Preeti Jain. Embedded Systems [Online]. Dosegljivo: http://www.engineersgarage.com/articles/embedded-systems [Dostopano: 25.5.2016].

[9] Miller, Robert. MIT OpenCourseWare, 6.831, UI Design and Implementation. 2004 [Online]. Dosegljivo: http://hdl.handle.net/1721.1/75810 [Dostopano: 30.5.2016].

[10] James L. Peterson, Theodore A. Norman. Buddy Systems, 1974 [Online]. Dosegljivo: ftp://www.cs.utexas.edu/pub/techreports/tr74-59.pdf [Dostopano: 17.5.2016].

[11] Vojtech Pavlik. Linux Input drivers v1.0 [Online]. Dosegljivo: https://www.kernel.org/doc/Documentation/input/input.txt [Dostopano 3.5.2016].

[12] Vojtech Pavlik, Hans de Goede. Input events [Online]. Dosegljivo: http://lxr.free-electrons.com/source/include/uapi/linux/input-event-codes.h. [Dostopano: 3.5.2016].

[13] Nicolas P. Rougier. Higher Quality 2D Text Rendering. Journal of Computer Graphics Techniques, 2013. [Online]. Dosegljivo: http://jcgt.org/published/0002/01/04/paper.pdf [Dostopano: 14.5.2016].

[14] Adafruit OLED displays [Online]. Dosegljivo: https://www.adafruit.com/category/98 [Dostopano: 13.5.2016].

[15] Businesswire Qt 5.2 [Online]. Dosegljivo: http://www.businesswire.com/news/home/20131212005484/en/Qt-5.2-launches-delivering-true-cross-platform-application [Dostopano 10. 5. 2016].

[16] Crystalfontz OLED display [Online]. Dosegljivo: https://www.crystalfontz.com/product/cfal12864lxy-128x64-module-oled-graphic [Dostopano: 13.5.2016].

[17] easyGUI application screen shots [Online]. Dosegljivo: http://www.easygui.com/easygui-application-screen-shots [Dostopano: 12.5.2016].

[18] Font rendering methods [Online]. Dosegljivo: http://www.benjamindumond.fr/pdf/Font%20rendering%20methods.pdf [Dostopano: 14.5.2016].

[19] fbset - Unix, Linux Command [Online]. Dosegljivo: http://www.tutorialspoint.com//unix_commands/fbset.htm [Dostopano 30. 4. 2016].

[20] FreeType [Online]. Dosegljivo: https://www.freetype.org [Dostopano: 22.5.2016].

[21] Pencil Project [Online]. Dosegljivo: http://pencil.evolus.vn [Dostopano: 17.5.2016].

[22] Rotary Encoder [Online]. Dosegljivo: http://www.tinkerforge.com/en/doc/Hardware/Bricklets/Rotary_Encoder.html [Dostopano: 10.6.2016].

[23] SEPS525 - Newhaven Display International, Inc [Online]. Dosegljivo: http://www.newhavendisplay.com/app_notes/SEPS525.pdf [Dostopano: 9.6.2016].

[24] Smart watch keyboard typing on Apple watch [Online]. Dosegljivo: http://www.smartwatchcrunch.com/smart-watch-keyboard-typing-on-apple-watch-android-wear [Dostopano: 17.5.2016].

[25] Simple Directmedia Layer [Online]. Dosegljivo: https://www.libsdl.org/index.php [Dostopano: 22.5.2016].

[26] True Type [Online]. Dosegljivo: https://www.microsoft.com/en-us/Typography/WhatIsTrueType.aspx [Dostopano: 22.5.2016].

[27] μGFX - a lightweight embedded library [Online]. Dosegljivo: http://ugfx.org/index.html [Dostopano: 12.5.2016].

[28] μGFX files [Online]. Dosegljivo: https://community.ugfx.org/files [Dostopano: 12.5.2016].

[29] μGFX demos [Online]. Dosegljivo: http://ugfx.org/demos.html [Dostopano: 12.5.2016].

Dodatek A

API programske knjižnice gradnikov uporabniškega vmesnika

V nadaljevanju so opisane ("set/get") metode razreda "widget", ki je glavni razred znotraj programske knjižnice gradnikov uporabniškega vmesnika. Je najpomembnejši, saj njegove metode podedujejo vsi ostali gradniki. Slednje tvorijo programski vmesnik, preko katerega lahko pridobivamo oziroma nastavljamo lastnosti posameznih gradnikov (barve, pisavo, položaj ...).

A.1 Razred widget

A.1.1 set_visible, get_visible:

void (*set_visible)(struct widget *widget, int visible);
int (*get_visible)(struct widget *widget);

Metodo set_visible() uporabimo za nastavljanje vidljivosti podanega gradnika. Slednji se ne bo izrisal, v kolikor bomo nastavili parameter visible na 0. Metodo get_visible() pa uporabimo, ko želimo ugotoviti, ali je gradnik

trenutno viden.

A.1.2 set_enabled, get_enabled:

void (*set_enabled)(struct widget *widget, int enabled);
int (*get_enabled)(struct widget *widget);

Metodo set_enabled() uporabimo za omogočanje in onemogočanje podanega gradnika. V kolikor bo onemogočen (0), se ne bo odzival na vhodne dogodke in bo izrisan v sivi barvi. Metodo get_enabled() pa uporabimo, ko želimo ugotoviti, ali je gradnik trenutno omogočen.

A.1.3 set_focus, get_focus:

void (*set_focus)(struct widget *widget, int focus);
int (*get_focus)(struct widget *widget);

Z metodo set_focus() lahko podanemu gradniku podelimo fokus (1) ali pa mu ga vzamemo (0). Slednji se odziva na vhodne dogodke le, če ima fokus. Ko želimo preveriti, če ima gradnik fokus, uporabimo metodo get_focus().

A.1.4 set_focusable, get_focusable:

void (*set_focusable)(struct widget *widget, int focusable);
int (*get_focusable)(struct widget *widget);

Z metodo set_focusable() določimo, ali je možno danemu gradniku podeliti fokus ali ne. Metodo get_focusable() uporabimo, ko želimo preveriti, ali lahko gradniku podelimo fokus.

A.1.5 set_location, get_location:

void (*set_location)(struct widget *widget, s32 x, s32 y);
void (*get_location)(struct widget *widget, wpoint_t *point);

Z metodo set_location() nastavimo lokacijo podanega gradnika na zaslonu. Pridobimo pa jo s klicem metode get_location().

A.1.6 set_size, get_size:

void (*set_size)(struct widget *widget, u16 w, u16 h);
void (*get_size)(struct widget *widget, wdim_t *dim);

Z metodo set_size() nastavimo velikost podanega gradnika. Pridobimo pa jo s klicem metode get_size().

A.1.7 set_bounds, get_bounds:

void (*set_bounds)(struct widget *widget, s32 x, s32 y, u16 w, u16 h);
void (*get_bounds)(struct widget *widget, wrect_t *bounds);

Z metodo set_bounds() nastavimo zunanje meje podanega gradnika (lokacijo in velikost). Pridobimo pa jih s klicem metode get_bounds().

A.1.8 set_margins, get_margins:

void (*set_margins)(struct widget *widget, u16 top, u16 bottom, u16 left, u16 right);
void (*get_margins)(struct widget *widget, wmargins_t *margins);

Z metodo set_margins() nastavimo odmike znotraj podanega gradnika (levi, desni, zgronji in spodnji odmik). Pridobimo pa jih s klicem metode get_margins().

A.1.9 set_border_width, get_border_width:

void (*set_border_width)(struct widget *widget, size_t width);
size_t (*get_border_width)(struct widget *widget);

Z metodo set_border_width() nastavimo širino zunanjega roba. Pridobimo pa jo s klicem metode get_margins().

A.1.10 set_color_*, get_color_*:

void (*set_bgcolor)(struct widget *widget, u32 bgcolor);
u32 (*get_bgcolor)(struct widget *widget);
void (*set_border_color)(struct widget *widget, u32 color);
u32 (*get_border_color)(struct widget *widget);
void (*set_text_color)(struct widget *widget, u32 color);
u32 (*get_text_color)(struct widget *widget);

Metode za nastavljanje in pridobivanje barv ozadja, zunanjega roba ter teksta.

A.1.11 set_font, get_font:

void (*set_font)(struct widget *widget, fbgfx_fontid_t fontid);
fbgfx_fontid_t (*get_font)(struct widget *widget);

Z metodo set_font() lahko nastavimo pisavo podanega gradnika. Pridobimo pa jo s klicem metode get_font().

A.1.12 set_css:

void (*set_css)(struct widget *widget, struct widget_css *css);

Z metodo set_css lahko nastavimo oblikovni stil podanega gradnika. Obli-

kovni stil je podan preko podatkovne strukture widget_css, ki vsebuje vse do sedaj opisane slogovne parametre.

A.1.13 process_input_event:

void (*process_input_event)(struct widget *widget, struct winput_event *ev);

Z metodo process_input_event() pošljemo vhodni dogodek podanemu gradniku. Gradnik se odzove tako, da podatek bodisi zavrže bodisi ga uporabi in spremeni svoj izris (npr. v vnosno polje se doda nov znak).

A.1.14 draw:

void (*draw)(struct widget *widget, struct fbgfx *gc);

Metoda za izris podanega gradnika.